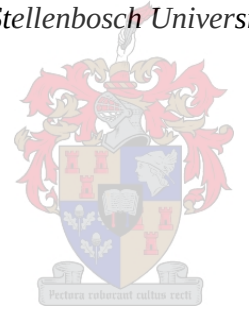
An ARTI Holonic Architecture Implementation for Table Grape Production Management

by
Johan Joubert Rossouw

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering Mechatronic in the Faculty of Engineering at Stellenbosch University



Supervisor: Dr Karel Kruger
Co-supervisor: Prof Anton Herman Basson

March 2021

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2021

Abstract

An ARTI holonic architecture implementation for table grape production management

J.J. Rossouw

Department of Mechanical and Mechatronic Engineering
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa
Thesis: MEng (Mechatronic Engineering)

March 2021

The management of table grape production is complex, since it must adhere to strict production requirements, facilitate numerous decisions by various experts, and integrate many human workers. With the ever increasing market demands, the table grape industry needs to adopt new technologies in order to stay competitive.

The objective of this thesis is to develop a system to aid the table grape production management. The developed system uses the Activity Resource Type Instance (ARTI) reference architecture to implement a holonic production management system, which is able to address the challenges of the table grape production management.

The ARTI components and their functionality are explained. The table grape production management is explained to identify the elements that makes up the production management. The mapping of the identified elements to the ARTI components is described in detail before the implementation of the system components are discussed.

The developed system is compared to the currently implemented system. The comparison is based on three key aspects of the table grape production manage system – communication, information management and decision support. The comparisons are drawn according to the functionality of each system to initiate, monitor, change and cancel a production order.

The results show that a production management system based on the ARTI reference architecture is able improve on the communication, information management and decision support of the currently implemented system. The developed system is also robust against disturbances and is flexible and adaptable to react to production order changes.

Uittreksel

‘n ARTI holoniese argitektuur implementering vir die bestuur van tafeldruifproduksie

J.J. Rossouw

Departement of Meganiese en Megatroniese Ingenieurswese
Universiteit Stellenbosch
Privaatsak X1, 7602 Matieland, Suid-Afrika
Tesis: MIng (Megatroniese Ingenieurswese)

March 2021

Die bestuur van tafeldruifproduksie is kompleks, omdat dit moet voldoen aan streng produksievereistes, talle besluite deur verskeie kundiges akkommodeer, en ‘n groot aantal werkers integreer. Die tafeldruifbedryf moet ook nuwe tegnologie implementeer om kompeterend te bly met die markvereistes wat aanhoudend toeneem.

Die doelwit van hierdie tesis is om ‘n stelsel te ontwikkel wat die bestuur van tafeldruifproduksie vergemaklik. Die ontwikkelde stelsel maak gebruik van die *Activity-Resource-Type-Instance* (ARTI) verwysingsargitektuur om ‘n holoniese produksiebestuurstelsel te implementeer wat die uitdagings in die bestuur van tafeldruifproduksie aanspreek.

Die ARTI komponente en hulle funksionaliteit word verduidelik. Die beheer van tafeldruifproduksiebestuur word verduidelik om die elemente te identifiseer waaruit die produksiebestuur bestaan. Die wyse waarop die elemente wat geïdentifiseer is met die ARTI komponente geassosieer word, word in detail bespreek voordat die implementering daarvan bespreek word.

Die ontwikkelde stelsel word vergelyk met die stelsel wat tans geïmplementeer is. Die vergelyking is gebaseer op drie kern aspekte van die produksiebestuurstelsel – kommunikasie, inligtingbestuur en besluitneming-ondersteuning. Die stelsels word vergelyk volgens die funksionaliteit van elke stelsel om ‘n produksiebestelling te skep, monitor, verander en te kanselleer.

Die resultate toon aan dat die produksiebestuurstelsel gebaseer op die ARTI verwysingsargitektuur op die stelsel wat tans geïmplementeer is, kan verbeter in terme van kommunikasie, inligtingbestuur en besluitneming-ondersteuning. Die ontwikkelde stelsel kan ook versteurings hanteer en is buigsaam en aanpasbaar om te reageer op produksie veranderinge.

Acknowledgements

I would like to thank Dr. Karel Kruger and Prof. Anton Basson for their guidance and expert advice during this thesis. Your willingness to share your knowledge and the aid you provided are much appreciated.

My father for providing me the opportunity and funding to pursue my ambitions. Also, providing in all my needs to complete this thesis

Nico Verster for his friendliness and willingness to aid with information about the table grape production management.

Carma Botha for her patience, love and support. You were always willing to listen and offer advice as best you could.

My family for the love, support and motivation provided, and faith they had in me.

My friends for their support and social events to lift my spirit and keep me going.

Above all, my heavenly Father for His blessings and talent He provided me with. Without Him nothing would be possible. He deserves all the honour and glory.

Table of contents

Page

List of figures ..... **xi**

List of tables ..... **xiv**

List of abbreviations ..... **xv**

1 Introduction **1**

1.1 Background 1

1.2 Objectives 3

1.3 Motivation 3

1.4 Methodology 4

1.5 Thesis Structure 5

2 Literature Review **6**

2.1 Holonic Systems 6

2.1.1 Theory of Holonic Systems 6

2.1.2 Holonic Manufacturing Systems 6

2.1.3 Holonic System Architectures 7

2.1.3.1 HCBA 7

2.1.3.2 ORCA-FMS 8

2.1.3.3 PROSA 8

2.1.3.4 ADACOR 9

2.1.3.5 Internal Architectures 10

2.1.3.6 Communication Protocols 11

2.2 ARTI Holonic Reference Architecture 13

2.3 Implementation Platforms for Holonic Architectures 15

2.4 ICT and IoT in Agricultural Applications 18

2.5 Discussion 20

3 Table Grape Production Management **21**

3.1 Assets and Facilities 21

3.1.1 Vineyards 21

3.1.2 Packhouses 21

3.1.3 Packing Material Storage Facility 22
3.2 Human Task Performers 22
3.2.1 Production Manager 22
3.2.2 Farm Manager 22
3.2.3 Packhouse Manager 23
3.2.4 Harvesting Teams 23
3.2.5 Quality Control Station Teams 23
3.2.6 Packing Station Teams 23
3.2.7 Transportation Vehicle Drivers 23
3.3 Production Processes 24
3.3.1 Grape Harvesting 24
3.3.2 Grape Quality Control 24
3.3.3 Grape Packing 24
3.3.4 Grape Transportation 25
3.3.5 Packing Materials Transportation 25

4 Conceptual Application of ARTI Architecture on Table Grape Production Management 26
4.1 Identification and Mapping of System Elements 26
4.2 Activities 28
4.2.1 Activity Type Intelligent Being 28
4.2.1.1 Production Order 29
4.2.1.2 Resource Selection 29
4.2.2 Activity Type Intelligent Agent 29
4.2.2.1 Production Order 29
4.2.2.2 Resource Selection 30
4.2.3 Activity Instance Intelligent Agent 30
4.2.4 Activity Instance Intelligent Being 30
4.3 Resources 30
4.3.1 Resource Type Intelligent Being 31
4.3.1.1 Packing Material Storage Facility 31
4.3.1.2 Production Manager 32
4.3.1.3 Farm Manager 32

4.3.1.4 Packhouse Manager 32
4.3.1.5 Component Manager 32
4.3.2 Resource Type Intelligent Agent 32
4.3.2.1 Asset and Facility Resources 33
4.3.2.2 Human Task Performer Resources 33
4.3.3 Resource Instance Intelligent Agent 33
4.3.4 Resource Instance Intelligent Being 33

5 ARTI Based Production Management System Implementation 34
5.1 Implementation Architecture 34
5.1.1 System Distribution 34
5.1.2 Software Platform 35
5.1.3 Interfaces 36
5.2 Holon Implementation 37
5.2.1 Implementation of ARTI Components 37
5.2.2 Activity Holon Implementation 39
5.2.2.1 Activity Type Intelligent Being 39
5.2.2.2 Activity Type Intelligent Agent 39
5.2.2.3 Activity Instance Intelligent Agent 41
5.2.2.4 Activity Instance Intelligent Being 41
5.2.3 Resource Holon Implementation 44
5.2.3.1 Resource Type Intelligent Being 44
5.2.3.2 Resource Type Intelligent Agent 48
5.2.3.3 Resource Instance Intelligent Agent 49
5.2.3.4 Resource Instance Intelligent Being 49
5.3 Holon Interactions 53
5.3.1 Initiate Production Order 53
5.3.2 Manage Production Order Resources 55
5.3.3 Terminate Production Order 55
5.3.4 Monitor Production Order 56
5.3.5 Managing Farm Resources 57
5.3.6 Previous Production Order Information 59

6 Evaluation 60
6.1 Evaluation Criteria 60
6.1.1 Communication 60
6.1.2 Information Management 61
6.1.3 Decision Support 62
6.2 Experiment Design 63
6.2.1 Initiate Production Order 64
6.2.2 Monitor Production Order 64
6.2.3 Production Order Changes 65
6.2.4 Cancel Production Order 66
6.3 Results and Discussion 66
6.3.1 Initiate Production Order 66
6.3.2 Monitor Production Order 68
6.3.3 Production Order Changes 72
6.3.4 Cancel Production Order 73
6.3.5 Results Discussion 74
6.3.5.1 Communication 74
6.3.5.2 Information Management 75
6.3.5.3 Decision Support 77
6.3.5.4 Discussion 79
6.4 Benefits and Drawbacks of the ARTI Holonic Architecture Implementation 79

7 Conclusion and Recommendations 81

8 References 83

Appendix A Tests Sniffer Screenshots 87
A.1 Initiate Production Order 87
A.2 Initiate Production Order with Limited Resources 91
A.3 Monitor Production Order 94
A.4 Production Order Changes 96
A.5 Cancel Production Order 100

Appendix B Messages in XML Format 101
B.1 Vineyard resource proposal information message in XML format 101

B.2 Request assigned resources inform message in XML format 101
B.3 Stored production order information in XML format 102

Appendix C JADE Source Code 103
C.1 Vineyard Resource 103

Appendix D Android Application Screenshots 123
D.1 Production Manager 123
D.2 Resources 125

List of figures

Page

Figure 1: Basic PROSA building blocks and information exchanged. (Van Brussel, et al., 1998) 8

Figure 2: ADACOR holon classes (Leitao, 2004).....10

Figure 3: Inter holon architecture (adapted from Foit, *et al.* (2017)). 11

Figure 4: Sequence diagram of the CNP (adapted from Bellifemine, *et al.* (2007)). 12

Figure 5: ARTI reference architecture cube. (Valckenaers, 2018).....13

Figure 6: Corrected ARTI reference architecture cube. (Valckenaers, 2020).....15

Figure 7: JADE split container (adapted from Bergenti, et al., (2014)). 18

Figure 8: Table grape PM decision levels.....27

Figure 9: Activity functionality mapping.....28

Figure 10: Resource functionality mapping.....31

Figure 11: Flow diagram of agent creation.....36

Figure 12: Flow of diagram NEU protocol.....38

Figure 13: Flow diagram of Android application Instance IB.....38

Figure 14: Flow diagram of a production order AIIB assigning new resources. 41

Figure 15: Flow diagram of a production order AIIB receiving a message. 42

Figure 16: Flow diagram of a resource selection AIIB selecting resources to assign to a production order.....44

Figure 17: Flow diagram of a resource handling production order assignments. 49

Figure 18: Flow diagram of a resource updating a production order task. 50

Figure 19: Flow diagram of a decision maker resource selecting resources. 51

Figure 20: Flow diagram of an Android application sending a message. 52

Figure 21: Flow diagram of a resource receiving a message. 52

Figure 22: Production order activity initiation and resource assignment sequence diagram 54

Figure 23: Sequence diagram of production manager adding and removing a resource from a production order. 55

Figure 24: Sequence diagram of a production manager terminating a production order.....56

Figure 25: Sequence diagram of a production manager monitoring a production order.....57

Figure 26: Sequence diagram of a farm manager creating and terminating a resource and acquiring and updating a resource. 58

Figure 27: Sequence diagram of a production manager acquiring the previous production orders and their information. 59

Figure 28: Sequence diagram for initiating a production order and assigning a resource. 67

Figure 29: Sequence diagram for requesting production order progress. 68

Figure 30: Sequence diagram for requesting assigned resources and assigned resource information. 69

Figure 31: Sequence diagram for updating progress and reaching target value. .69

Figure 32: Sequence diagram of the packhouse manager updating the packaged pallets, reaching the target and completing the production order....70

Figure 33: Sequence diagram for requesting previous production orders and a production order's information. 71

Figure 34: Sequence diagram of the farm manager requesting farm resources, requesting a resource's information and updating the resource's information. 71

Figure 35: Sequence diagram of the production manager removing a resource from a production order and obtaining a new list of assigned resources. 72

Figure 36: Sequence diagram of a resource removing itself from a production order.....72

Figure 37: Sequence diagram of the production manager adding a resource to a production order.....73

Figure 38: Sequence diagram for cancelling a production order. 74

Figure 39: Initiate production order first set of messages. 87

Figure 40: Initiate production order second set of messages. 88

Figure 41: Initiate production order third set of messages. 89

Figure 42: Initiate production order final set of messages. 90

Figure 43: Initiate production order with limited resources first set of messages. 91

Figure 44: Initiate production order with limited resources second set of messages. 92

Figure 45: Initiate production order with limited resources final set of messages. 93

Figure 46: Production manager requesting a production order's assigned resources, an assigned resource's information and a production order's progress update. 94

Figure 47: Farm manager retrieving its farm's resources, a resource's information and updating a resource's information. 95

Figure 48: Request available resources and remove a vineyard resource from the production order. 96

Figure 49: Replace a harvesting team resource. 97

Figure 50: Add a grape transportation vehicle. 98

Figure 51: Add a packing material transportation vehicle. 99

Figure 52: Cancel production order. 100

Figure 53: Production manager home screen (a) and production manager screen to view a vineyard resource (b). 123

Figure 54: Production manager resource selection screen (a) and production manager resource addition screen (b). 124

Figure 55: Harvesting team resource before adding 5 crates to the TEST production order (a) and after adding 5 crates (b). 125

List of tables

Page

Table 1: ATIB behaviours for a production order activity 39
Table 2: ATIA methods for a production order activity 40
Table 3: ATIA methods for a resource selection activity 40
Table 4: AIIA methods for a production order and resource selection activity 41
Table 5: SSIteratedAchieveREResponder behaviours to respond to messages 43
Table 6: RTIB methods for resources running on a PC 45
Table 7: RTIB methods for resources running on a mobile device 45
Table 8: RTIB methods for a production manager resource 46
Table 9: RTIB methods for a farm manager resource.....47
Table 10: RTIB methods for a packhouse manager resource 47
Table 11: RTIA methods of the resources.....48
Table 12: RIIA methods for a production order and resource selection activity ..49
Table 13: SSIteratedAchieveREResponder behaviours to respond to messages ..53
Table 14: Predetermined resource sets for experiments. 63
Table 15: Communication accuracy results 74
Table 16: Communication reliability results 75
Table 17: Information management accessibility results.....75
Table 18: Information management traceability results 76
Table 19: Decision support consistency results 77
Table 20: Decision support agility results 78

List of abbreviations

ADACOR	ADAptive holonic COntrol aRchitecture
AIIA	Activity Instance Intelligent Agent
AIIB	Activity Instance Intelligent Being
AOP	Agent-orientated programming
ATIA	Activity Type Intelligent Agent
ATIB	Activity Type Intelligent Being
CFP	Call For Proposal
CNP	Contract Net Protocol
CS	Current System
DS	Developed System
DT	Digital Twin
FB	Function Block
FSM	Finite-State Machine
GUI	Graphical User Interface
HCBA	Holonic Component-Based Architecture
HMS	Holonic Manufacturing System
IA	Intelligent Agent
IB	Intelligent Being
ICT	Information and Communication Technology
IoT	Internet of Things
JADE	Java Agent Development
LEAP	Lightweight and Extensible Agent Platform
MAD	Mechatronics Automation and Design
MAS	Multi-Agent Systems
NEU	Next Execute Update
O2A	Object-to-Agent
PC	Personal Computer
PM	Production Management

PROSA	Product Resource Order Staff holons Architecture
RIIA	Resource Instance Intelligent Agent
RIIB	Resource Instance Intelligent Being
RTIA	Resource Type Intelligent Agent
RTIB	Resource Type Intelligent Being

1 Introduction

This chapter introduces and provides the background of the thesis, followed by the objectives, motivation and methodology. Finally, an overview of the structure of the thesis is provided.

1.1 Background

The fourth industrial revolution has brought forth the evolution of cyber-physical systems made possible through the developments in the IT infrastructure. These developments include the increased usage of the Internet to wirelessly connect resources, information, objects and people with each other, to create the Internet of Things (IoT). The IoT has been adopted in the manufacturing industry to improve their efficiency in order to stay competitive. (Kagermann, et al., 2013)

The table grape industry is still very labour intensive. The reason is that table grapes are a very perishable product and must be handled with care while they are being harvested, pruned and packaged. To automate this process is extremely difficult and expensive, due to the delicate manner required when handling grapes to avoid damaging them. However, the fourth industrial revolution enables the development of new technologies in order to stay competitive by improving efficiency and productivity. (Sihlobo, 2019)

With the introduction of these new technologies, more complex systems are bound to develop. This will also lead to dynamic systems, providing new and complicated challenges to implement and control these systems. The demand for a control system capable of adapting to these frequent and sudden changes will continue to increase.

In the table grape industry, it is already a common practice to use technology in a cooperative manner, such as electronic scales being used to weigh the packaged grapes, as well as measure the grape packing efficiency. Besides from the technologies already being used, new technologies are being researched such as remote soil moisture sensors with weather station data to improve water management (Rossello, et al., 2019), pest bird detection, classification and recognition using deep learning (Bhusal, et al., 2019), fruit grading systems using machine vision (Xie, et al., 2019) and navigation system using a real-time image-based system for precise driving (Lin & Chen, 2019).

The incorporation of these technologies requires that the personnel have sufficient training on how to use them. Frequently monitoring the information these technologies provide can thus be time consuming. The fourth industrial revolution will produce even more technologies, which will lead to more complex systems. This trend will lead to production systems that are too complex for a single person to manage whilst maintaining focus on growing the grapes and keeping the vineyards healthy.

One way to overcome this challenge is by assigning a person whose only responsibility is to monitor and manage these production systems, but this can be an expensive solution. Another solution is to have a Production Management (PM) system that simplifies the management process and reduces the time consumed of supervisors to ensure the PM is done correctly.

The PM of a table grape farm includes all the aspects from when an order is received, until the finalized order is ready to leave the farm. A typical process to complete such an order consists of harvesting grapes from vineyards and transporting the harvested grapes and packing materials to a packhouse where the quality of the grapes is inspected before packing the grapes in the packing materials. Managing all these aspects can become time consuming to ensure each step in the production process is executed correctly and on time. A PM system to reduce the complexity and time consumed by the table grape production processes must include the management of all these aspects.

Holonic systems – discussed in section 2.1 – are constructed from autonomous and cooperative entities used as building blocks to model complex systems. The Activity Resource Type Instance (ARTI) reference architecture – discussed in section 2.2 – specifies that these autonomous and cooperative entities be divided into activities, resources, types and instances. The ARTI reference architecture can be used to implement the holonic architecture, which could be a feasible solution for the table grape PM.

The Mechatronics Automation and Design (MAD) research group at Stellenbosch University focuses on researching the fourth industrial revolution and holonic systems. Holonic systems can be used to simplify and model complex systems. The MAD research group have explored the use of holonic systems in the manufacturing domain. Developing a table grape PM system based on the ARTI reference architecture is the first research in the group to explore the use of ARTI and the use of holonic systems beyond the manufacturing domain.

1.2 Objectives

The objective of this thesis is to develop a PM system for table grapes based on the ARTI holonic reference architecture. The development of the PM system will consist of the design and synthesis of the system according to the table grape PM requirements. The scope of the thesis will include all the PM aspects during the harvesting season; from when an order is received, until an order is completed. The project will only consider the PM changes as the orders received from the export companies change. The aspects that need to be included are the management of the packing material stock and transportation thereof, the harvesting and transportation of the grapes, and the packing of the grapes in the packhouses. The scope will not include the management stages following the completion of an order. The implementation will be limited to a single farm with a single packhouse; however, the project will make provision for the addition of farms and packhouses.

1.3 Motivation

With the continuous development of technology, the world seems to become smaller as everything becomes more connected. In the table grape industry this can be both beneficial and challenging. It is beneficial in the way that the communication between the market, the export company and the farmer has improved and, therefore, the market demands are better known and more easily met. This creates additional challenges in terms of managing the farms to keep up with sudden market demand changes.

Due to improved technologies and communications systems, the time it takes for a farmer to be affected by sudden changes in market demands are reduced. Additionally, the South African farmers are not only competing against farmers from within their own region, but also with farmers in different countries such as Australia, Peru, Chile and Brazil that can be much closer to the market. Being close to the market reduces the time it takes for grapes to reach the markets, which reduces the time required to adapt to market changes. Producing the best possible quality grapes, and adapting as quickly as possible to market changes, is crucial to remain competitive.

Currently, the table grape industry is very dependent on human labour to accomplish most of the tasks. During the packing season, a farm of about 40 hectares can require up to 200 workers per day (Rossouw, 2019). The farm manager must thus spend time supervising each individual to ensure that all the necessary tasks are completed on time and with required quality. Furthermore, the packing orders can be changed frequently and suddenly throughout the day.

A sudden change can be time consuming and expensive when everyone affected by the change must be informed and receive new instructions. A farm of about 40 hectares can produce up to 170 000 boxes of grapes per season (Rossouw, 2019). This causes great challenges in the management of the packing materials, grapes to be harvested and the people to execute the tasks. Having a PM system that handles not only the changes of the packing orders, but also the execution thereof can simplify the complexity of the process and make it less time consuming. Having a single distributed system controlling the production of the whole farm can also reduce the mistakes made by communication and the execution of tasks.

Holonic systems improve stability towards disturbances and is adaptable and flexible towards changes (Van Brussel, et al., 1998). Therefore, a holonic system should be capable of managing production order changes efficiently and effectively while being robust against PM disturbances. The holonic systems architecture divides a system into autonomous and cooperative entities to reduce the complexity of a system and create a flexible and adaptable system to handle frequent and sudden changes. The ARTI reference architecture uses generic terminology, making it applicable beyond the manufacturing domain.

1.4 Methodology

The PM system will be based on the ARTI holonic architecture. This requires that all table grape PM tasks be translated and modelled with the holonic architecture. The ARTI reference architecture is used to perform this translation by dividing each of the table grape PM aspects into activities, resource, types and instances.

The scope of the project is selected to include as many of the table grape PM aspects as possible, while maintaining focus and only including the aspects that provide value to the purpose of the project. The scope is restricted to a single farm while making provision for the addition of farms. All the PM aspects of the single farm will be included to cover all the aspects adding value to the table grape PM.

The chosen case study is a table grape farm. Most of the PM aspects on the farm is done by manual labour with the help of tools, such as vehicles to transport grapes and packing materials. The farms do not have sensors to gather information on the table grape production aspects and their progress. The current system (CS) relies on supervisors to provide information on the production and knowledgeable personnel to make decisions and give instructions.

Thorough understanding of holonic systems and the ARTI reference architecture was required to implement the table grape PM system. The literature on holonic systems and the ARTI reference architecture was studied and is presented in chapter 2. This provided insight into the implementation of such a system, as well as the elements that should be identified in the table grape PM system.

The PM system should be designed specifically with the requirements and needs of a table grape farm in mind. Therefore, it is important to understand each task of a table grape farm and how it fits into the PM. A table grape farm was observed to identify all the relevant aspects of the table grape farm that form part of the table grape PM and is presented in chapter 3.

All the identified aspects that are relevant to the table grape PM must be mapped according to the ARTI specifications to create an ARTI-based system. The mapping of table grape PM aspects to the ARTI components are presented in chapter 4.

The platform chosen to implement the table grape PM system must be able to implement the system according to the mapping of the table grape PM to the ARTI components. The platform must also be able to implement all the necessary functionality to fulfil the requirements of such a system. The implementation of the system on the chosen platform is presented in chapter 5.

To evaluate the developed system (DS), evaluation criteria were constructed and experiments were designed accordingly. The experiments were conducted with both systems. The evaluation criteria are then used to compare the CS and DS and discuss the results of both systems in chapter 6.

1.5 Thesis Structure

Chapter 2 presents a review of the relevant literature. The relevant literature includes holonic systems, the ARTI reference architecture, implementation platforms, agricultural Information and Communication Technology (ICT) and IoT. Finally, the findings from the literature review are discussed.

Chapter 3 describes the table grape PM, focussing on the assets and facilities, human task performers and the production processes.

Chapter 4 describes the conceptual application of ARTI on the table grape PM and describes the identification of the system elements that must be implemented according to the ARTI specifications.

Chapter 5 describes the implementation of the ARTI-based table grape PM system, focussing on the implementation of each activity and resource holon after providing an overview of the implementation architecture. Finally, the interactions between the system holons are discussed.

Chapter 6 discusses the evaluation of the DS by first defining the evaluation criteria and the experiments, before discussing the results.

Finally, Chapter 7 presents the conclusions drawn from this thesis, as well as the recommendations for future work.

2 Literature Review

This chapter reviews the literature on holonic systems and the implementation thereof. The theory of holonic systems, holonic manufacturing systems and holonic system implementation architectures are discussed in section 2.1. Sections 2.2 and 2.3 follow with a discussion on the ARTI reference architecture and the platforms to implement the holonic architecture. Section 2.4 provides a discussion on ICT applications in agriculture. Finally, section 2.5 offers a discussion on the reviewed literature.

2.1 Holonic Systems

2.1.1 Theory of Holonic Systems

The holonic systems approach originates from the theories of Arthur Koestler (Koestler, 1967). The word 'holon' consists of the Greek word holos meaning *whole* and the suffix 'on' meaning a particle or *part*. Holons are individual entities that can be a part of larger entities, while simultaneously consisting of numerous autonomous and cooperative entities. Koestler prosed this word 'holon' to describe the hybrid nature of these systems consisting of parts and wholes.

These holons form autonomous and cooperative building blocks to model complex systems. They are autonomous in the sense that each holon is an entity that can create its own plans and/or control the execution thereof. They are cooperative in the sense they can develop mutually acceptable plans and/or strategies and execute them. The holons contains an information and physical processing part which allows them to transport, store and/or validate information and physical objects in systems. Altogether the holons in the holonic system cooperate to achieve a desired goal or outcome. (Van Brussel, et al., 1998)

2.1.2 Holonic Manufacturing Systems

Holonic Manufacturing Systems (HMS) is a paradigm that aims to take advantage of the autonomous and cooperative characteristics of holons to create a flexible system to address the changes and uncertainties in manufacturing (Singgih, 2014). The application of HMS provides the advantages of considering the dynamic manufacturing environments, modelling and investigating the interactions between manufacturing components and organizing manufacturing and computational entities as system components (Xie & Liu, 2017).

HMS use holons to reduce the complexity of systems by breaking it down into entities representing a single system task. These entities mirror the part of reality they represent. The holons group together to form clusters, forming an aggregation hierarchy. The holons within the aggregation hierarchy can belong to multiple aggregations depending on the functionality of the holons and where the system requires these holons. The aggregated holons can be designed up front or they can be created dynamically by interactions with other holons. The aggregation holons can also dynamically change to adapt to the desires of the system. (Van Brussel, et al., 1998)

Heterarchical structures produce robust systems, but are myopic and do not provide performance optimization. Hierarchical structures produce systems with good predictability and performance optimization at the expense of robustness. The aggregation of holons form a hybrid structure called a *holarchy*, which inherit the benefits from both heterarchical and hierarchical systems. The holarchies enable the holons to cooperate and combine their knowledge and skills to achieve complex system goals. This enables easy reconfiguration, extension and modification of the system resulting in a more flexibility and adaptable system. (Van Brussel, et al., 1998)

The HMS specifies that the system components be divided into holons. However, the HMS do not specify how these system components must be mapped to holons. Therefore, several holonic reference architectures have been proposed to specify the mapping of system components, namely: HCBA (Chirn & McFarlane, 2000), ORCA (Pach, et al., 2014), ADACOR (Leitao, 2004) and PROSA (Van Brussel, et al., 1998)

2.1.3 Holonic System Architectures

2.1.3.1 HCBA

The Holonic Component-Based Architecture (HCBA) categorises the physical manufacturing plant objects into resources and products according to their properties. Resources contain the properties of performing manufacturing operations, where products contain the properties to accept manufacturing treatments. Both the resources and products are mapped to system holons.

The resources consist of a physical processing part representing the physical manufacturing process and an information processing part responsible for the control, communication and decision making of the manufacturing process. Similarly, the products consist of a physical processing part representing the raw materials or manufacturing parts and an information processing part responsible for the routing control, decision making and process control. (Chirn & McFarlane, 2000)

2.1.3.2 ORCA-FMS

The Optimized and Reactive Control Architecture (ORCA) - Flexible Manufacturing System (FMS) is a hybrid holonic reference architecture that is able to dynamically switch between a hierarchical control and a heterarchical control. The hierarchical control is a predictive mode that is executed when the system behaves as planned. The heterarchical control is a reactive mode that is executed whenever an event occurs that prevents the planned behaviour of the system to be executed. (Pach, et al., 2014)

2.1.3.3 PROSA

The Product-Resource-Order-Staff Architecture (PROSA) requires that a system be dived into three basic holons, called resource holons, product holons and order holons. The resource holons contains the information of physical parts that can be used in a manufacturing process, as well as the information on how resources can be used in production processes. The product holons contain all the information on the final products of the manufacturing processes and communicates this information to the other holons. The order holons are used to divide the manufacturing processes into tasks. Each task is represented by an order holon and contains the information on how each task should be executed. (Van Brussel, et al., 1998)

Throughout the manufacturing process, these holons cooperate to achieve system goals. Figure 1 shows the basic holons of PROSA, as well the information sent between the respective holons. Product holons and resource holons communicate by sending process knowledge. The process knowledge is the information on how a process should be performed given a specific resource. Product holons and order holons communicate by sending production knowledge. The production knowledge is the information on how a product should be produced given the available resources and processes. Resource holons and order holons communicate by sending process execution knowledge. The process execution knowledge is the information on the progress that is made by a process to produce a product from a resource. (Van Brussel, et al., 1998)

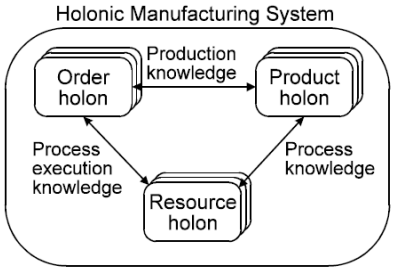


Figure 1: Basic PROSA building blocks and information exchanged. (Van Brussel, et al., 1998)

Additional holons can be added to the holonic manufacturing system, called staff holons. These staff holons contain information that can be used by the basic holons to assist them with performing their functions. The staff holons only assist the basic holons by providing additional information, but the basic holons still make the decisions. The staff holon is considered an expert that gives advice and is used to achieve global optimization. The staff holons together with the basic holons form the PROSA architecture. (Van Brussel *et al.*, 1998)

PROSA was originally developed for manufacturing systems. ARTI was introduced to change the terminology of the PROSA holons to be generic and abstract. This allowed for an opportunity to improve the mirroring of reality of the reference architecture and to make it applicable beyond the manufacturing domain (Valckenaers, 2020). In ARTI, the order holons are replaced with activity instances, the product holons with activity types, and the resource holons are replaced and subdivided into instances and types (Valckenaers, 2018). The ARTI reference architecture is discussed in section 2.2.

2.1.3.4 ADACOR

The ADAptive holonic Control aRchitecture (ADACOR) is designed for distributed manufacturing systems, focussing on the planning and control of the production at the shop floor level. ADACOR also breaks down the manufacturing control functions into autonomous and cooperative entities called holons. This enables the system to inherit the advantages of the modularity, decentralisation, agility, flexibility, robustness and scalability of the holons. This enables ADACOR to have agile reaction to frequent occurring disturbances creating an agile and flexible control system. (Leitao, 2004)

The ADACOR architecture divides the holons into product (ProdH), task (TH), operational (OpH) and supervisor (SupH) holon classes. These holons are based on the functions and objectives of the components within a manufacturing factory. These holons classes are represented in Figure 2, showing the class and its responsibility within the manufacturing system. (Leitao, 2004)

The product holons are similar to the product holons in PROSA and represent the products which the manufacturing system can produce. The product holons contain the knowledge about the products and is therefore responsible for the short-term process planning, scheduling and execution. (Leitao, 2004)

The task holons are similar to the order holons in PROSA and represents a production order. The task holon is responsible of managing the shop floor execution of a product and contains the information about the order. (Leitao, 2004)

The operational holons are similar to the resource holons in PROSA and represents the physical shop floor resources, each with their own goals and skills, the system can use to accomplish tasks. (Leitao, 2004)

The supervisor holon is added to the manufacturing system to introduce global optimization and coordination of resources. The supervision holon optimizes production plans and dynamically group holons together to combine their skills and to provide a combined service to evolve according to the changing environment. (Leitao, 2004)

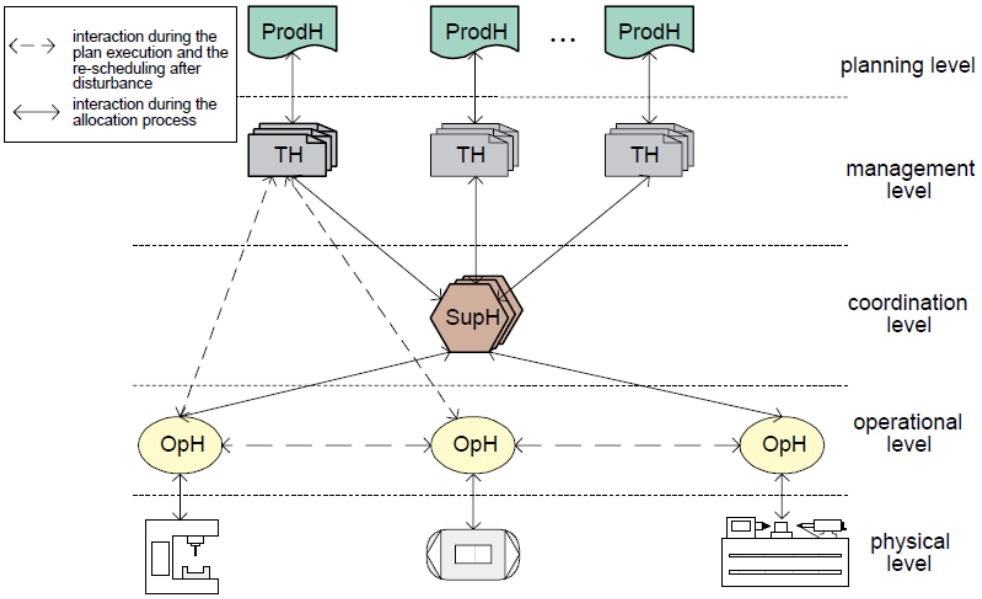


Figure 2: ADACOR holon classes (Leitao, 2004).

2.1.3.5 Internal Architectures

The general architecture of a holon contains a physical and an information processing part. The physical processing part is the actual hardware used to perform operations and is controlled by a physical controller. The information processing part is used to make decisions and consists of two interfaces: an interface that allows holons to interact with each other and an interface to allow holons and humans to interact with each other. (Bussmann, 1998)

Figure 3 shows the internal architecture of a holon containing an information and physical processing part, as formulated by Foit, *et al.* (2017). The information processing part allows for autonomy, and together with the communication network, cooperation can be achieved. The internal interface between the information and physical processing allows the information processing part of the holon to optimize the control of the physical execution.

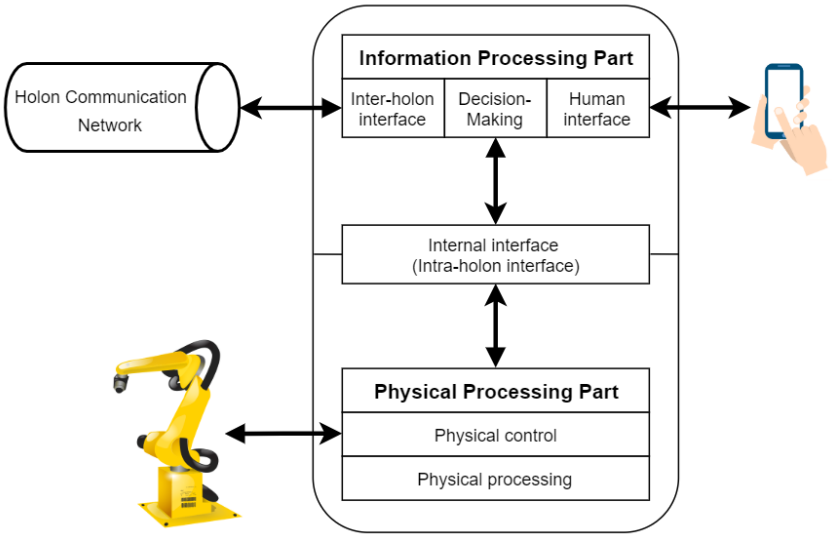


Figure 3: Inter holon architecture (adapted from Foit, *et al.* (2017)).

2.1.3.6 Communication Protocols

The Next-Execute-Update (NEU) protocol enables holonic systems to cope with unexpected changes. The NEU protocol functions as an interactive help desk which decouples technical aspects from execution aspects. The interactive help desk knows all possible execution sequences to achieve a system goal. This allows system elements to request that the interactive help desk computes which operations to execute next. The system elements can continue with execution and reality reflection while the interactive help desk determines the state to which the system elements should change. This allows either the technical aspects or the execution aspect to evolve without having to change both simultaneously. (Valckenaers & De Mazière, 2015)

The Contract Net Protocol (CNP) is a communication protocol that allows a system to perform decentralized task allocations though dynamic negotiations of contracts (Xu & Weigand, 2001). The CNP also facilitates cooperation by distributing the control of the execution of tasks. Each agent in the system can either be a manager or a contractor. The managers monitor the execution of tasks, whereas the contractors are responsible for the execution of these tasks. Any agent can dynamically change between a manager or contractor (Smith, 1980). Figure 4 shows the sequence diagram for a typical CNP.

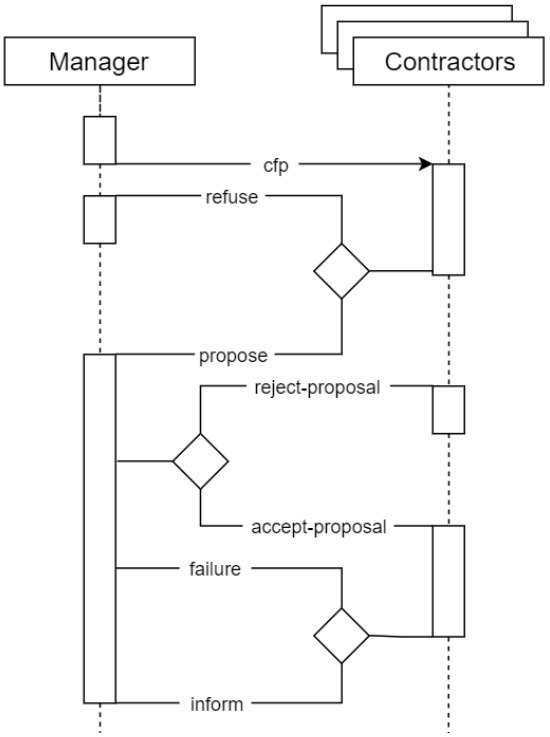


Figure 4: Sequence diagram of the CNP (adapted from Bellifemine, *et al.* (2007)).

The CNP consist of three stages: an announcing stage, bidding stage, and awarding stage. In the announcing stage, the manager agent sends out a Call For Proposal (CFP) to contractors that are able to perform the required task. The CFP contains the task specifications and any other conditions. In the bidding stage, the contractors who are able to perform the task send a proposal containing the preconditions of the contractor, such as completion time or cost. The contractors who are unable to perform the task refuse the CFP. The manager agent then chooses the contractors according to their proposals using a special algorithm. In the awarding stage, the manager sends an *accept-proposal* to the chosen contractors and a *reject-proposal* to the contractors that were not chosen by the manager. Finally, the contractors can respond by informing the manager if the task failed, the task is complete or send the result of the task.

2.2 ARTI Holonic Reference Architecture

The ARTI reference architecture was created when PROSA was reconsidered and uses the holonic systems approach to simplify the modelling and control of complex systems. ARTI addresses the shortcomings of PROSA, and proposes more generic terminology, to offer support to applications beyond the manufacturing domain.

The ARTI reference architecture divides a complex system into a collection of holons classified into three dimensions (as shown in Figure 5): Activity or Resource, Type or Instance, and Intelligent Being or Intelligent Agent. The Intelligent Agents (IA) are represented with green cubes and the Intelligent Beings (IB) with the blue cubes. The way in which the IA interacts with the IB depends on the way they are connected to the IB. These connections are represented by the yellow cubes. The IAs therefore function as staff holons by giving advice.

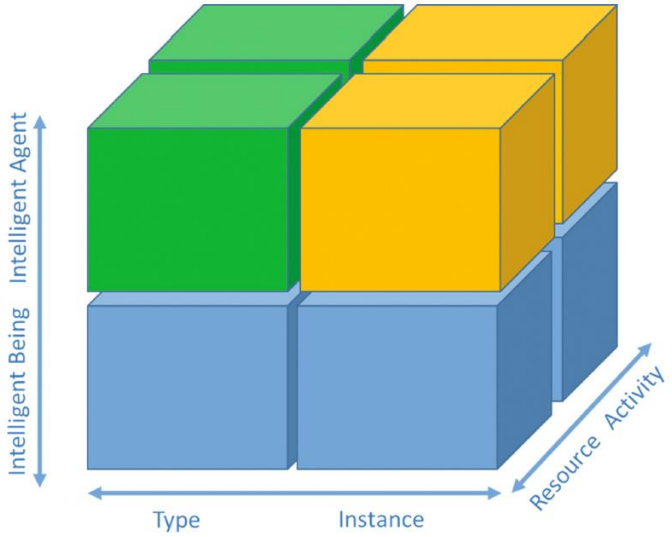


Figure 5: ARTI reference architecture cube. (Valckenaers, 2018)

ARTI prescribes that the holons in the system can either be Resources or Activities – holons can either perform some service, or coordinate the performance of services by other holons. Furthermore, a holon can be classified as a Type or an Instance. Type holons contain the expert knowledge and functionality to support the performance of system tasks, while Instance holons are responsible for performing system tasks. Finally, a holon can either be an IB or an IA. The IB holons can reflect and affect the state of the real or virtual system, and are thus capable of performing system tasks. The IA holons encapsulate the decision-making functionality required for the effective performance of system tasks.

Since the ARTI architecture maps a complex system to several holons of specific type and functionality, each holon becomes responsible for managing and monitoring its own small environment. Having each holon monitoring and managing only a small part of the system reduces the overall complexity and increases the stability of the system. Furthermore, the architecture allows for easy modification by only having to add or remove single holons instead of changing entire system sections and their interactions with the rest of the system.

For the ARTI reference architecture to sufficiently simplify a complex system while maintaining flexibility and modification, the architecture requires that each system component be mapped into a single ARTI cube. Since many systems are still strongly dependent on humans, the ARTI architecture makes provision for humans performing system tasks. However, humans are inherently equipped with abilities that allow them to span across multiple ARTI-cubes and can therefore not be encapsulated in a single cube. Therefore, humans are represented as activity performers performing tasks spanning multiple cubes. (Valckenaers & Van Brussel, 2015)

Valckenaers (2020) reported that after exploring ARTI applications in domains beyond manufacturing, the ARTI cube was corrected, and visualized more precisely, leading to the ARTI cube in Figure 6. The IB components still describe the mirroring of the world-of-interest and reflect reality. The IA components correspond to the decision making to achieve system goals. ARTI distinguishes between the decision making IA components from the IB components mirroring their real-world counterparts to preserve the aggregation and specialization achieved with PROSA. The IB components are now referred to as Digital Twins (DTs) of their corresponding real-world counterpart. This generalizes the wording of IB referring to a component mirroring a real-world counterpart. The use of DTs improves the communication and transfer of knowledge and avoids confusing situations.

The exploration of applications beyond the manufacturing domain further refined the green elements, consisting of decision-making tools and the yellow elements connecting the blue elements to the green elements. The boundary between the decision making elements and blue elements has been refined. The blue elements consist of the reality-reflecting components. This includes reflecting the physical real-world counterpart, as well as the decision-making counterpart. The blue cubes thus consist of the physical DT as well as the decision-making DT. The decision-making DT instances allow the system to virtually execute the intentions of the decisions using the activities and resources. The green decision-making tool elements are available to the blue decision-making DT elements to be used whenever required, using the yellow elements as link between them. (Valckenaers, 2020)

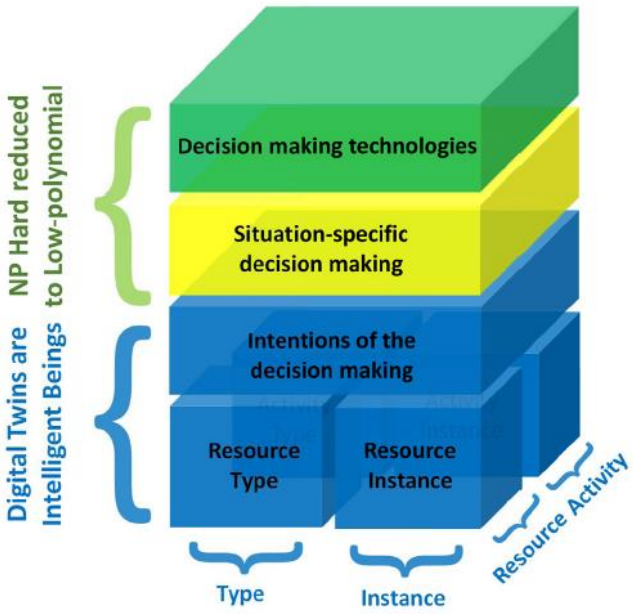


Figure 6: Corrected ARTI reference architecture cube. (Valckenaers, 2020)

The representation of a real-world counterpart in the cyber space is considered a DT. The DT provides the advantages of visualization, collaboration and decision making enhancement (Redelinghuys, et al., 2020). Implementing a DT based on the ARTI reference architecture was recently explored by Borangiu, *et al.* (2020) and Cardin, *et al*. (2020). The IB components are used to fulfil role of mirroring a real-world counterpart. The IA components contain the skills and knowledge to change the behaviour of the DT represented by the IB components Borangiu, *et al*., 2020).

2.3 Implementation Platforms for Holonic Architectures

Various implementation platforms exist for the implementation of holonic architectures, such as IEC 61499 Function Blocks (Kruger & Basson, 2013), Erlang (Kruger & Basson, 2016) and C# (Kotze, 2016). However, this section will focus on the Java Agent Development (JADE) framework. JADE is one of the most commonly used development tools to implement an agent-orientated system ((Meng, et al., 2006), (Lin & Chen, 2019) and (Kruger & Basson, 2018)). JADE was originally developed by the Research and Development department of Telecom Italia, but since 2000 it is available under an open source license. JADE is a written completely in Java, which allows developers to create Multi-Agent Systems (MAS) with all the benefits from the Java language features and third party libraries. This allows developers to easily construct a JADE multi-agent system with relatively little expertise in the theory of agents (Bellifemine, et al., 2007).

Agent-Orientated Programming (AOP) is used to create a system consisting of autonomous, proactive entities, called agents, with the ability to communicate to one another. Autonomy allows the agent to independently perform complex tasks that take up some time. Proactiveness allows the agents to initiate and perform tasks without the input from a user. The agents are able to communicate allowing them to work together in a cooperative manner to achieve their own goals and the goals of other agents within the system. (Bellifemine, et al., 2007)

Agents are similar to the concept of objects and share properties such as encapsulation and message passing. However, agents differ from objects in the sense that they are autonomous and are capable of flexible tasks since each agent has its own thread. In JADE, these tasks which an agent can perform are implemented as agent behaviours. These behaviours must be added to the agent to make the agent execute them. Behaviours can be added to an agent at any time or they can be added from within other behaviours. (Bellifemine, et al., 2007)

Agent-based systems allows the development of flexible control system that can adapt to changing production conditions. Agent-based systems are also inherently adaptable and reconfigurable due to the inherit characteristics of the agents (Meng, et al., 2006). Since these characteristics are also required by holons and holonic architectures, as discussed in section 2.1.3, it makes agent-based systems a popular choice to implement holonic architectures. Since JADE is an implementation platform for agents, JADE is a popular choice for implementing holonic architectures.

The agent management framework within which the agents can exist consists of an agent platform (AP), a directory facilitator (DF), an agent management system (AMS) and a message transport system (MTS). The AP is the physical infrastructure where the agents are deployed. The agents are the computational processes that inhabit the AP. The DF maintains a list of agents and provides the agents with a service of finding agents with specific services. The AMS is responsible to manage the AP. This includes creating and terminating agents as well as migrating agents to and from an AP. The MTS provides the service to transport messages between agents within the same or different APs. (Bellifemine, et al., 2007)

The communication protocols used by JADE agents comply with the Foundation for Intelligent, Physical Agents (FIPA) specifications for agent communication. This requires that messages conversations are managed through predefined actions, or communication acts, while different content languages can be used. Communication acts that are most commonly used are inform, request, agree, not understood and refuse. These communication acts form the basis of most conversations. (Bellifemine, et al., 2007)

Multi-agent application can become quite complex and can be distributed across multiple hosts. This implies difficulties in managing and debugging such a system. JADE provides a JADE Remote Monitoring Agent (RMA), which is a graphical management console through which other tools can be launched to ease the agent management and system debugging. The DummyAgent is used to simulate sending messages to test the behaviours of other agents. The Sniffer agent subscribes to the AMS and is notified of all events and message exchanges between agents. The Introspector agent is used to debug behaviours within a single agent. (Bellifemine, et al., 2007)

The JADE framework in which agents can be implemented unfortunately cannot run on small devices for the following reasons (Caire & Pieri, 2011):

1. The JADE runtime environment requires more memory than that which handheld devices offer due to the limitations of these devices.

2. Not all handheld devices support Java 5 or later, as is required by JADE.

3. Fixed and wireless network links have different characteristics which needs to be taken into account.

To solve these problems, the Lightweight and Extensible Agent Platform (LEAP) add-on was created. LEAP enable the deployment of JADE agents on handheld devices. This is achieved by some part of the JADE kernel forming a modified runtime environment to create JADE powered by LEAP (JADE-LEAP) that can be deployed on a wide range of handheld devices. (Caire & Pieri, 2011)

Two execution modes exist with the JADE runtime environment: stand-alone execution mode and split execution mode. With stand-alone execution mode, the complete container where the JADE runtime is activated runs on the device. With the split execution mode, the container where the JADE runtime is activated is split into a front-end and a back-end. The front-end runs on the device running the JADE runtime. The back-end runs on a remote server that is permanently linked to the device running the JADE runtime. This setup is shown is shown in Figure 7. The split execution mode is more suitable for devices with a limited capacity, since the front-end is more lightweight. The back-end, referred to as the mediator, can then be used to perform the heavy workloads. Multiple mediators can also be deployed to distribute the workload. (Caire & Pieri, 2011)

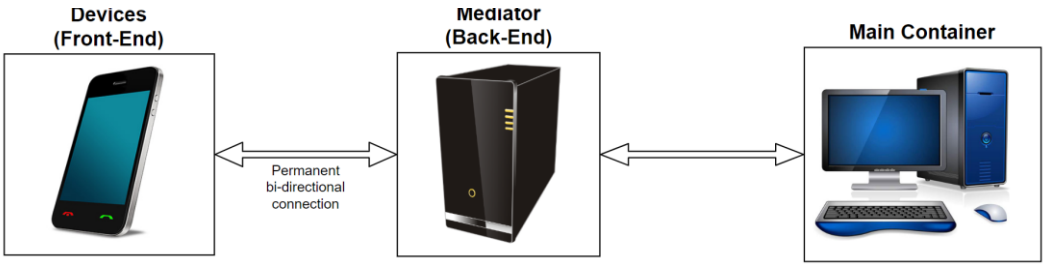


Figure 7: JADE split container (adapted from Bergenti, et al., (2014)).

Version 4.1.1 of JADE introduced the Object-to-Agent (O2A) interface mechanism to pass information from Graphical User Interface (GUI) components to agents. This mechanism allows the GUI components on an Android device to pass information to an agent, enabling communication to the agent. This allows the user, or an external component, to trigger behaviours within agents. The agent is also able to communicate with the user, or external components, with a mechanism provided by Android that allows different application components to interact with one another. This mechanism uses *intents*, which is based on broadcasting the information that can be received by any component interested in the intent. (Bergenti, et al., 2014)

2.4 ICT and IoT in Agricultural Applications

In most cases, the management of a farm is based on the farmer's experience and knowledge instead of real-time information about the farm. This makes it difficult to manage a farm while overcoming the challenges regarding efficient use of resources and reducing environmental impacts while continuing to make a profit. (Perea, et al., 2017)

In the agricultural industry is can be difficult to find knowledgeable aid. IoT systems can provide farmers with information that can be used to improve their knowledge about their farms and improve the management thereof. However, systems based on IoT are not always reliable due to unreliable network connections. (Mohanraj, et al., 2016)

ICT can be used in agriculture to provide farmers with accurate information whenever the farmers requires the information (Mahant, et al., 2012). Examples of where ICT can be used are provided in Aqeel-ur-Rehman, *et al*. (2014). Using wireless networks, ICT applications were created to assist in:

- Irrigation with weather data and wireless soil moisture sensors.
- Fertilization with sensors to acquire real-time soil data.
- Pest control by monitoring humidity and temperature to prevent diseases on vegetables.
- Animal and pastures monitoring by means of sensors to monitor animal behaviour.
- Horticulture by using sensors to monitor the environment of nurseries for optimal growth.

ICT is also used to send weather forecasts obtained from satellite data to farmers in Amarnath, *et al*. (2018). The best results were achieved by using an SMS service to send a summary of the forecast information to farmers.

ICT can thus provide benefits in agriculture; however, agriculture can only benefit from ICT when they overcome the challenges regarding the quality and availability of network coverage, easy and affordable accessibility, and the willingness to adopt new technologies. (Mahant, et al., 2012)

The following aspects define an open-air engineering process according to Valckenaers & Van Brussel (2015):

- Management of mobile equipment such, as vehicles.
- Interaction with non-flat surfaces in 3D space.
- Removal and deposit of materials.
- Logistics to transport materials

This means that an open-air engineering system must not only efficiently adapt to frequent and unexpected production changes, but also to changes produced by the changing work performance of tasks, cooperation between workers, co-operation between successive system tasks, changes in environmental conditions and changes induced by market demands. The system needs to exhibit agility in response to changes and robustness in its handling of disturbances. (Ali, et al., 2012)

2.5 Discussion

Holonic systems can be used to reduce the complexity of a complex system to create a flexible and adaptable system. PROSA has been thoroughly explored and evaluated as an architecture to implement holonic systems in the manufacturing domain. ARTI proposes a more generic architecture to assist in the implementation of a holonic system beyond the manufacturing domain. However, ARTI is relatively new and has not yet been thoroughly explored and evaluated. Therefore, there is not a great deal of literature available on ARTI. The research on ARTI has been mostly done by Paul Valckenaers, who proposed the PROSA and ARTI reference architecture, and others mentioned in section 2.2.

The table grape industry can be defined as an open-air engineering process (as discussed in section 2.4), since it is dependent on mobile equipment, such as vehicles, to transport grapes and packing materials, it interacts with non-flat surfaces in a 3D space, grapes and packing materials are removed and deposited, and it contains logistics by transporting grapes and packing materials. This induces challenges with regards to ICT and IoT in agriculture, such as the availability of quality network coverage. However, some solutions have been developed and have dealt with these challenges.

A Holonic architecture can provide a platform to develop a distributed system consisting of multiple autonomous and cooperative entities to aid in the research to address the challenges in agriculture. Each entity mirrors a small part of the reality, which reduces the complexity of the system. Additionally, this can improve the adaptability of the system by having multiple entities adapting to their small part of reality they represent. The ARTI reference architecture uses generic wording to specify how a system must be broken down into autonomous and cooperative entities to develop a holonic system.

JADE and JADE-LEAP can be used to implement a distributed holonic system based on the ARTI reference architecture. JADE provides tools that eases the development of a MAS. JADE also contains communication protocols, such as the NEU and CNP discussed in section 2.1.3.6, that can be used to ease the development of a holonic system.

3 Table Grape Production Management

Table grape PM is a process of managing the grapes from when they are harvested until they are packaged. The production is executed according to production orders. Production orders contain all specifications regarding the grapes, the packing process, packhouse accreditation and registrations, and the packing materials to be used.

This chapter will discuss the assets and facilities used in completing a production order in section 3.1, followed by the task performers to execute the tasks required by the production order in section 3.2. Finally, insight will be provided in section 3.3 of the processes that must be managed to complete a production order.

3.1 Assets and Facilities

3.1.1 Vineyards

The vineyards produce the grapes that are used in the production process. The vineyards are grown on farms divided into orchards. Each of these orchards contains a variety of grapes. This allows a single farm to execute a variety of production orders. The vineyards are classified according to the variety and quality of the grapes. The quality of the grapes is determined according to their colour, sugar level, blemish, size of the berries and the number of grapes they produce. When a new production order is initiated, this will be used to determine which vineyard to assign to satisfy the production order requirements. (Kritzinger, 2020)

3.1.2 Packhouses

The packhouses are the buildings where the quality of the grapes is inspected and the grapes packaged. Each farm has its own packhouse located at a central location on the farm. The packhouse provides an environment where the temperature and humidity are regulated. Cooler temperatures and a more humid atmosphere during the packing process increase the shelf life of the grapes. Each packhouse is classified according to the throughput capacity of grapes, the markets for which the packhouse is registered to package grapes for, and the food safety and hygiene accreditation of the packhouse (as required by certain markets). (South African Department of Agriculture, Forestry and Fisheries, 2012)

3.1.3 Packing Material Storage Facility

The packing material storage facility is a building located at a central location to all the farms of the company. The packing material storage facility stores the packing materials until they are required by a production order. The packing materials consist of the inner packaging in which the grapes are packaged, carton boxes in which the inner packaging is placed, labels placed on the carton boxes to identify the grapes, and sulphur dioxide sheets to prevent fungal growth during storage and transportation of the packaged grapes (Star South, 2019).

3.2 Human Task Performers

3.2.1 Production Manager

The main responsibility of the production manager is initiating, managing and cancelling production orders. The production manager negotiates to establish production orders with the grape export companies. The production manager is also responsible for negotiating production order changes.

The production manager is also responsible to assign packing materials to production orders, as well as allocate the vineyard where the grapes should be harvested and the packhouses where the grapes should be packaged. Multiple vineyards and packhouses may be assigned to a single production order, or multiple production orders to a single packhouse or vineyard.

3.2.2 Farm Manager

The main responsibility of the farm manager is to manage a farm. Each farm consists of vineyards, a packhouse containing quality control stations and packing stations, vehicles and workers. The workers are allocated by the farm manager to specific tasks according to their abilities and skills. The workers perform the harvesting of the grapes, the transportation of grapes and packing materials using the farm's vehicles, grape quality control and grape packing.

The farm manager will assign workers to harvesting teams to harvest the grapes according to the vineyard allocated to the production order. The quality control stations and packing stations will be assigned with workers according to the number of available workers and the rate at which the production order must be executed. The farm manager also allocates the farm's vehicles to transport grapes and packing materials according to the assigned vineyards and packing materials.

3.2.3 Packhouse Manager

The main responsibility packhouse manager is to manage the packhouse. This is done by managing the quality control stations by ensuring that the grapes are correctly inspected and managing the packing stations by ensuring that the correct packing materials are used during the grape packing.

3.2.4 Harvesting Teams

Harvesting teams consist of a team of workers. The workers are assigned to specific teams according to their skills and abilities. The number of workers per team is chosen by the farm manager according to the availability of workers and the rate at which the harvesting must be done. The harvesting teams cut the grapes from the vineyards and place them in plastic crates.

3.2.5 Quality Control Station Teams

The quality control station teams consist of a table with a group of assigned workers. The workers are allocated to a station by the farm manager according to their abilities and skills. The stations are located next to a conveyor belt that feeds the crates of harvested grapes into the packhouse. The quality control station teams inspect the quality of these grapes before they are packaged.

3.2.6 Packing Station Teams

The packing station teams consist of a table with a group of assigned workers. The workers are allocated to a station by the farm manager according to their abilities and skills. The stations are located next to a conveyor belt that feeds the grapes that satisfy the production order's quality specifications to the packing stations. The packing station teams pack the grapes before they are ready to leave the packhouse.

3.2.7 Transportation Vehicle Drivers

The transportation vehicle drivers can be a driver of one of three vehicle types: tractors, trucks and pick-up trucks. The tractors are mainly used to transport the grapes from the vineyards to the packhouses, but can be equipped with trailers to perform different tasks. The trucks are mainly used to transport large loads such as packaged grapes. The pick-up trucks are mainly used to transport small numbers of workers and packing materials. The farm manager also uses them to move around to and inspect the farm. The vehicles can also be assigned to tasks other than their main purpose whenever the availability of the vehicles is limited.

3.3 Production Processes

3.3.1 Grape Harvesting

Harvesting is done by the harvesting teams using scissors to cut the grapes from the vines. The grapes are then placed into plastic crates. These plastic crates can then be loaded onto a vehicle to transport the grapes to a packhouse. Harvesting teams perform the harvesting to handle the grapes with care and to prevent damaging them while cutting the grapes from difficult to reach places.

The production manager selects the vineyard according to the production order specifications and informs the farm manager by sending an SMS or email. The farm manager then selects the harvesting teams to perform the harvesting and verbally gives the teams their instructions. The harvesting progress is obtained by verbally requesting the harvesting information from the harvesting team supervisor.

3.3.2 Grape Quality Control

The packhouse contains a precooler where the harvested grapes are delivered. The grapes are then cooled down after being in the hot outside temperatures before entering the packhouse. The grapes then pass through a quality control station where the quality of the grapes are inspected and the berries that are damaged, contain blemish, are too small or did not colour enough are removed by cutting them out using scissors.

The production manager selects the packhouse according to the production order specifications and informs the farm manager by sending an SMS or email. The farm manager then selects the quality control stations to perform the quality control and verbally gives the stations their instructions. The quality control progress is estimated from the number of packaged grapes.

3.3.3 Grape Packing

The grapes that satisfy the production order's quality specification pass though the packing stations where the they are packaged. The packing stations pack the grapes by first placing the grapes in either plastic boxes, called *punnets*, or plastic bags. These *punnets* or bags differ with regards to their size to contain weights of grapes, ranging from 500g to 1500g, and are placed within an inner packaging plastic bag before they are placed within carton boxes. Sulphur dioxide sheets are then placed in the carton boxes to prevent fungal growth during the grape storage and transportation. Finally, the boxes are closed and the labels specifying the grapes and target market are pasted onto the boxes.

The grape packing is done in the same packhouse chosen by the production manager for the quality control. The farm manager selects the packing stations to perform the packing and verbally gives the stations their instructions. The grape packing progress is obtained by comparing the number of packaged grapes to the production order target value.

3.3.4 Grape Transportation

The grapes that are harvested and placed in plastic crates need to be transported to the packhouse where the grapes are packaged. Usually, tractors with trailers will be allocated to perform the grape transportation. The tractors will have a flatbed trailer and are able to drive through the vineyards. The plastic crates containing the grapes are then stacked on the trailer. Once the trailer is full, the tractor transports the grapes to the relevant packhouse before repeating its journey until all the grapes have been transported.

The vehicles to transport the harvested grapes are selected by the farm manager. The farm manager verbally gives the grape transportation instructions to the driver. The grape transportation progress is obtained by requesting the progress from the driver either verbally or with an SMS.

3.3.5 Packing Materials Transportation

The packing materials are ordered and arrive before the start of a production order. The packing material storage facility allows the company to distribute and manage the packing materials across multiple farms. This is especially useful when multiple production orders are executed simultaneously or when a single production order is executed on multiple farms in multiple packhouses. The packing materials are usually transported using either the trucks or pick-up trucks – depending on the quantity of packing materials. The packing materials are stacked onto the vehicle and transported to the relevant packhouse. The vehicle will repeat its journey, if necessary, until all the packing materials are transported.

The vehicles to transport the packing materials are selected by the farm manager. The farm manager verbally gives the packing material transportation instructions to the driver. The packing material transportation progress is obtained by requesting the progress from the driver, either verbally or with an SMS.

4 Conceptual Application of ARTI Architecture on Table Grape Production Management

This chapter will discuss the conceptual implementation to create an ARTI reference architecture application for table grape PM. Firstly, section 4.1 provides a discussion on the identification of the system elements and mapping thereof according to the activities and resources from the ARTI cube in Figure 5. Section 4.2 discusses the mapping of each activity's components to ARTI cubes. Finally, section 4.3 discusses the mapping of each resource's components to ARTI cubes.

4.1 Identification and Mapping of System Elements

The application of the ARTI reference architecture requires that the table grape production assets and facilities, human task performers and processes, discussed in section 3, be mapped to the ARTI cubes shown in Figure 5 in section 2.2. This requires that each task be classified as either an activity, that coordinates services, or a resource that performs a service.

The resources identified in the table grape PM, discussed in section 3, are vineyards, packhouses, a packing material storage facility, a production manager, farm managers, packhouse managers, harvesting teams, quality control stations, packing stations and transportation vehicles. These resources all perform some service in the production management.

The production and farm manager resources represent the production and farm managers. The assignment of resources is complex, since there are numerous variables involved. To automate the assignment of resources will require an advanced algorithm to ensure global optimization of the production processes. Currently, the production and farm managers ensure that the most suitable resources are assigned using their experience and expertise. Since the resource assignment is complex, it is desired to retain the decision-making functions of the production and farm managers.

The packhouse managers are used to update the packhouse resources. The other identified resources perform some service for the production orders they are assigned to. In addition to the identified resources, a component manager is added as a resource. The component manager encapsulates the functionality to create new system components, such as resources or activities. This simplifies the implementation of creating components in the system. The functionality of the component manager and the necessity thereof will be discussed in section 5.2.3.1.6.

The services that an activity must coordinate must be provided by resources. Therefore, resources must be assigned to each activity. One way to approach the identification of activities is to classify the production processes, discussed in section 3.3, as activities.

One of the complexities of assigning resources are where two activities must share the same resource. For example, the quality control and grape packing activities are performed within the same packhouse. Both the quality control and packing activities require a packhouse to be assigned to them. Therefore, the packhouse first needs to be assigned to a production order before the quality control and packing activities can assign quality control stations and packing stations.

Another complex resource assignment is the grape transportation activity requiring both vineyard and packhouse resources. The vineyard resources are assigned to the grape harvesting activity and the packhouses to the quality control and grape packing activities. This creates a complex interaction between activities to keep track of the resources that need to be assigned to activities and the resources already assigned to previous activities that are dependent on the same resource. Since the production and farm managers decide which resources to assign, the system will need to coordinate what resources are already assigned to previous activities and what resources still need to be assigned.

Since the decision-making functionality is retained by the production and farm managers, the identification of activities can alternatively be approached by classifying the activities according to the two levels of decision making, as shown in Figure 8. The production manager makes decisions regarding the vineyards, the packhouses and the packing material to assign to a production order. Accordingly, the farm manager managing the farm of the assigned vineyard chooses the harvesting teams. The farm manager managing the farm of the assigned packhouse chooses the packing and quality control stations. The farm managers will also be responsible for choosing vehicles from the farm's transportation fleet for the grape and packing material transportation, depending on which farms the assigned vineyard and packhouse resources are located.

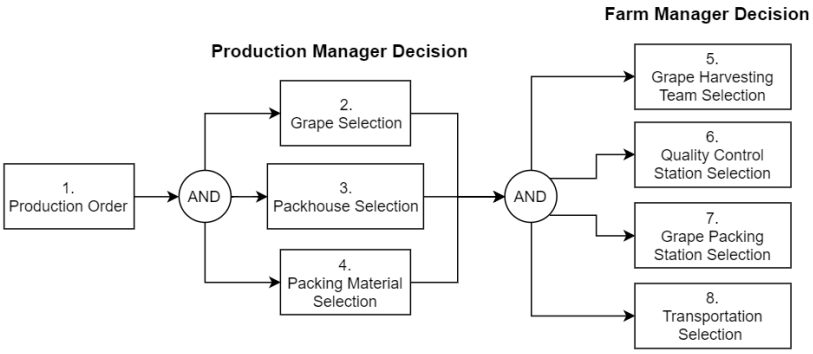


Figure 8: Table grape PM decision levels.

The production order needs to keep track of the progress of each of its tasks. This can either be done by obtaining the progress from the activities coordinating production order tasks, or the production order can acquire the progress of its tasks directly from the resources. Therefore, the production order does not necessarily require individual activities to coordinate its tasks. The use of activities to coordinate individual services for a production order adds to the complexity of the system without adding enough value to the system to justify the complexity thereof. Therefore, the activities chosen for the implementation of the system consists of a production order activity and a resource selection activity according to the two levels of decision making. The resource selection activities are used to assign resources according to the two levels of decision making.

4.2 Activities

From section 4.1, the identified activities that need to be implemented are the production order and resource selection activities. This section will discuss the mapping of each activity to the ARTI components with reference to Figure 9. The four ARTI component are: Activity Type Intelligent Being (ATIB), Activity Type Intelligent Being (ATIB), Activity Instance Intelligent Agent (AIIA) and Activity Instance Intelligent Being (AIIB).

Figure 9: Activity functionality mapping.

4.2.1 Activity Type Intelligent Being

The ATIB contains the activity-specific execution functionalities. Since the production order and resource selection activities contain different execution functionalities, they will be discussed individually.

4.2.1.1 Production Order

The production order ATIB encapsulate the functionality to:

- Initiate a new activity to select resources for the production order.
- Update the production order as resources make progress towards the production order.
- Keep an updated list of the resources assigned to the production order.
- Store the information about the production order.

4.2.1.2 Resource Selection

The resource selection activity is initiated for each resource type, as discussed in section 4.1. Although the execution of the resource selection activities differs according to the resource type the activity is initiated for, they require the same execution functionality. Therefore, the resource selection activities can use the same ATIB. Alternatively, the resource selection activity for each different resource type can have its own ATIB component. This alternative is preferred since it makes provision for the addition of a selection activity that may require different execution functionalities.

The resource selection ATIB encapsulate the functionality to:

- Discover the resources available to be assigned to a production order.
- Obtain information on the available resources to determine their suitability to the production order.
- Request that a decision maker assign resources to the production order according to the information obtained on the available resources.

4.2.2 Activity Type Intelligent Agent

The Activity Type Intelligent Agent (ATIA) contains the activity-specific decision-making functionalities. Since the production order and resource selection activities contain different decision-making functionalities, they will be discussed individually.

4.2.2.1 Production Order

The production order ATIA contains the functionality to specify:

- The ATIB from which the AIIB must obtain its execution functionality.
- What type of resources to assign to the production order.
- When the production order resource assignment is complete.
- How to respond to different messages received by the production order activity.
- What execution functionality the AIIB should execute next depending on the result of the previously executed functionality.

4.2.2.2 Resource Selection

Similar to the ATIB components of the resource selection activities, the ATIA components for the different resource selection activities contain the same decision-making functionality, although the outcome of their decisions differs. Therefore, the resource selection activities can use the same ATIA. Alternatively, the resource selection activity for each different resource type can have its own ATIA component. Again, this alternative is preferred since it makes provision for the addition of a selection activity that may require different decision-making functionalities.

The resource selection ATIA contains the functionality to specify:

- From which ATIB the AIIB must obtain its execution functionality.
- The decision maker to assign resources to the production order, according to the specific type of resource.

4.2.3 Activity Instance Intelligent Agent

The AIIA component links the AIIB to the correct Activity Type Intelligent Agent ATIA where the decision making takes place. Therefore, the AIIA is a generic component that applies to all activities. Since the AIIB component is generic, the system can be updated to accommodate a new activity by independently adding a new ATIB component. The same generic AIIB will be initiated and its execution will be determined by its corresponding ATIB component.

4.2.4 Activity Instance Intelligent Being

The AIIB is a generic component for both the production order and resource selection activities. The AIIB represents the real-world activity in the virtual system and only contains the functionality to perform the Next Execute Update (NEU) protocol. The NEU protocol enables the AIIB component to obtain the activity-specific task to execute from the ATIB, to execute the task, and to obtain the next task to execute depending on the result of the previous task.

4.3 Resources

From section 4.1, the identified resources that need to be implemented are the vineyards, packhouses, packing material storage facility, harvesting teams, quality control stations, packing stations, vehicles, production manager, farm managers, packhouse managers and the component manager. This section will discuss the mapping of each resource to the ARTI components with reference to Figure 10. The four ARTI component are: Resource Type Intelligent Being (RTIB), Resource Type Intelligent Being (RTIB), Resource Instance Intelligent Agent (RIIA) and Resource Instance Intelligent Being (RIIB).

Figure 10: Resource functionality mapping.

4.3.1 Resource Type Intelligent Being

The RTIB contains the resource-specific execution functionalities. Although the purpose of each resource differs, most of the resources require the same functionality. All the resources contain the functionality to receive, and respond to, messages received from activities and other resources. As identified in section 4.1, the resources that do not contain the same functionality are the packing material storage facility, production manager, farm managers, packhouse managers and component manager. These resources will be discussed individually.

The RTIB components of most of the resources encapsulate the execution functionality to:

- Handle requests to assign the resource to a production order.
- Update the information of the resource.
- Update the production order with the resources latest progress.
- Manage the schedule of the resource with assigned production orders.

4.3.1.1 Packing Material Storage Facility

The packing material storage facility RTIB encapsulate the functionality to:

- Handle requests to assign packing materials to a production order.
- Update the packing material stock in the storage facility when new stock arrives or when stock is collected to be used in a production order.

4.3.1.2 Production Manager

The production manager RTIB contains the functionality to:

- Create a new production order.
- Make changes to an existing production order.
- Cancel an existing production order.
- Select resources to assign to a production order.
- Request information on the active production orders and their progress.
- Request information on the resources assigned to a production order and their progress toward the production order.
- Retrieve the previous production orders and their information.

4.3.1.3 Farm Manager

The farm manager RTIB contains the functionality to:

- Create or remove vineyard or packhouse resources on a farm.
- Update the information of a vineyard or packhouse resource, e.g. the quality of a vineyard's grapes or the accreditation of a packhouse.
- Select resources to assign to a production order.
- Request information on the active resources on a farm.

4.3.1.4 Packhouse Manager

The packhouse manager RTIB contains the functionality to update the information of a packhouse resource when its information changes or when it makes progress towards a production order.

4.3.1.5 Component Manager

The component manager RTIB contains the functionality to:

- Create a new production order activity.
- Create a new vineyard resource.
- Create a new packhouse resource.
- Retrieve previous production orders and their information.

4.3.2 Resource Type Intelligent Agent

The RTIA contains the resource-specific decision-making functionality to make decisions regarding the execution functionality in the RTIB. Although the RTIB components of the resources differ, there are still common decision-making functionalities. The RTIA components of all the resources specifies the RTIB from which the RIIB must obtain its execution functionality and to specific how to respond to messages. The RTIA component of the component manager, as identified in section 4.1, only contains these functionalities.

4.3.2.1 Asset and Facility Resources

The RTIA components of the vineyard and packhouse asset and facility resources, as identified in section 4.1, contains the same decision-making functionality. The packing material storage facility RTIA component contains the same functionality as the human task performer resources, as motivated in section 5.1.1 and discussed in section 4.3.4.2.

The vineyard and packhouse RTIA components contain the functionality to specify:

- How to handle requests to assign the resource to a production order.
- How to store the information of the resource.
- What functionality to execute next depending on the result of the previous functionality.
- How to update the schedule of the resource.

4.3.2.2 Human Task Performer Resources

The RTIA components of the human task performer resources, as identified in section 4.1, and the packing material storage facility contain the same decision-making functionality, except for the production and farm managers. The RTIA components contain the functionality, that is common of all these resources, to specify:

- How to handle requests to assign the resource to a production order.
- How the resource information should be stored.
- How to update the schedule of the resource.

The production and farm manager RTIA components contain the above-mentioned functionality, as well as an additional functionality to specify how to handle requests to assign a specific resource type to a production order.

4.3.3 Resource Instance Intelligent Agent

Similar to the AIIA component, the RIIA component is a generic component that applies to all resources and links the RIIB to the correct RTIA where the resource specific decision making takes place. Since the RIIB component is generic, the system can be updated to accommodate a new resource by independently adding a new RTIB component to determine the execution of the same generic RIIB.

4.3.4 Resource Instance Intelligent Being

Similar to the AIIB, the RIIB component represents the real-world resources in the virtual system and is generic so that it can be used for all the types of resources. The RIIB performs the NEU protocol enabling the RIIB component to obtain the resource-specific task to execute from the RTIB, to execute the task, and to obtain the next task to execute depending on the result of the previous task.

5 ARTI Based Production Management System Implementation

This chapter discusses the implementation of the ARTI-based PM system. The chapter starts with an overview of the implementation architecture, providing a broad implementation overview, followed by a discussion of the implementation of the holon entities, before finally discussing the interaction between the holons to achieve the desired system functionality.

5.1 Implementation Architecture

This section provides an overview of the implementation. Firstly, an overview of the distributed system is provided, followed by a discussion on the software used for the implementation of the distributed system and the hardware considerations to achieve the distribution. Finally, the interfaces between the different system components are explained.

5.1.1 System Distribution

By nature, the PM is distributed across the entire farm. Therefore, the ARTI based PM system needs to be distributed to include all the activities and resources. The production order and resource selection activities run on a Personal Computer (PC). The production order activities are initiated on request from a production manager. The selection activity agents are initiated by production order agents when it requires resources to be assigned to a production order.

The resources are represented by agents running on either a PC or a mobile device. The vineyards, packhouses and component manager agents run on a PC, since they are more passive in the PM as described in section 4.1. The packing material storage facility agent runs on a mobile device together with the human task performer resources: harvesting teams, quality control stations, packing stations, vehicles, production manager, farm managers and packhouse managers.

The human task performer resources are considered to be more active in the PM, as described in section 4.1. They have the ability to move around on the farm and need to update their information dynamically. These resources are accessed via mobile devices to provide the functionality that allows them to adapt to the dynamic, distributed PM environment by enabling them to easily update their information dynamically.

Network coverage on farms, as discussed in section 2.5, is not always available. Mobile devices running an Android application have a cellular network to give them internet access where there is no WiFi. Therefore, phones can be used to create a distributed system on the farm. Phones are also commonly available and easy to operate. Therefore, phones are a practical and cost efficient solution to achieve distribution.

The resource agents running on mobile devices are initiated and available for the system to use when a user logs into the application. When the user logs out, the agent is terminated. The resource agents running on a PC are created by either a production or farm manager. These resources stay active in the system until they receive a request from a production or farm manager to terminate.

5.1.2 Software Platform

JADE was chosen to implement the activities and resources running on a PC. As mentioned in section 2.3.1, JADE can be used to develop a system consisting of autonomous, proactive entities called agents with the ability to communicate to one another allowing them to work together in a cooperative manner.

JADE is a written completely in Java and provides the benefits from the Java language features and third-party libraries, easing the development of a GUI that enables the system to interact with users. JADE also provides a DF that maintains a list of agents and provides the agents with a service of finding agents with specific services. The MTS of JADE provides the service of transporting FIPA messages between agents within the system to ease the implementation of agent communication and agent cooperation. JADE also provides graphical tools, such as the DummyAgent, Sniffer and Introspector agents discussed in section 2.3.2, to support development and debugging.

JADE-LEAP is chosen to implement the resources running on mobile devices. As discussed in section 2.3.2, JADE-LEAP is a lightweight version of JADE and can be used to implement an agent based system across multiple devices. The split execution mode provides a lightweight front-end that can run on a device with a limited capacity, while the back-end runs on a mediator container, such as a PC, to handle the heavy workloads. This makes the system more versatile.

JADE also introduced the O2A interface mechanism to pass information from a GUI to the agents. This enables Android applications to pass information to an agent, allowing a user to trigger behaviours within agents. JADE and JADE-LEAP also provide the agents with the ability to communicate with an application with intents, which is based on broadcasting the information to be received by any interested application component.

5.1.3 Interfaces

JADE-LEAP is used to create the agents on mobile devices that need to interface with the users and with the rest of the system running on a PC using JADE. The interface between JADE-LEAP on the mobile devices and JADE on the PC is facilitated by JADE. The interface between agents and the user consists of the O2A interface mechanism and broadcasting intents, as described in section 2.3.2.

Figure 11 shows the sequence of events to create an agent on a mobile device. When a user logs into the Android application, the agent is created accordingly. First, the JADE runtime must be activated by binding the MicroRuntimeService to the Android application to enable the agent functionalities. Next, a container needs to be created to host the agent. With split execution mode, the container is created on a mediator server.

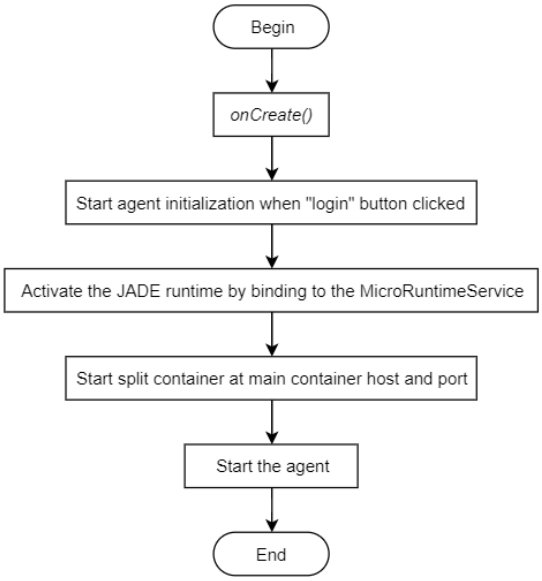


Figure 11: Flow diagram of agent creation.

The mediator server can be the same as the PC hosting the rest of the JADE agents of the system, or any other server, depending on the specified IP address and communication port of the host server running JADE. Once the container is successfully created, the agent can be launched in the container on the mediator server. The back-end hosting the agent is permanently linked to the mobile device running the JADE runtime using JADE-LEAP.

Once the agent is created, the user can interact with the system through interactions with the agent. When the Android application receives an input from the user, the O2A interface mechanism calls a method within the agent. The method can either add a behaviour to the agent or just execute a method.

When the agent needs to send information to the user, the agent will broadcast an intent that can be received by any interested application component that has registered to receive the type of intent. The intent consists of an action to identify the intent and the content of the intent.

5.2 Holon Implementation

The functionality of the activities and resources are mapped according to chapter 4. To achieve this functionality, the implementation uses a combination of methods and behaviours. This section discusses the methods and behaviours required by each activity and resource to achieve these functionalities.

5.2.1 Implementation of ARTI Components

The Instance and Type IA components of both the activities and resources encapsulate the decision-making functionality. In this application, the decision making is retained in decision-making resources, such as the production and farm managers. Therefore, the IA components are implemented as classes, since they do not require the functionality offered by agents to perform complex decision-making processes or algorithms.

The Instance IB components of both the activities and resources are implemented and instantiated as agents that do all the execution, by executing behaviours in a dedicated operating system thread. The functionality which they execute is obtained from the Type IB components. The Type IB components of both the activities and resources encapsulate the activity or resource specific execution functionality. Although they contain the execution functionality, the Type IB components are implemented as classes. This allows the Instance IB to execute the NEU protocol – discussed in section 2.1.3.6 and presented in Figure 12 – by instantiating the behaviours obtained from the Type IB. Similarly, the Instance IB also uses method calls to instantiate the objects from the IA component classes.

The NEU protocol presented in Figure 12 shows how the Instance IB uses the Type IA to specify the behaviour to execute. Throughout section 5.2, the legend used within the figures indicates where the functionality executed in the Instance IB is obtained. The Instance IB obtains the Type IA from the Instance IA, before obtaining the Type IB from the Type IA. The Type IA also specifies the first behaviour the to execute. The Instance IB obtains the behaviour to execute from the specified Type IB. Once the Instance IB executed a behaviour, the result of the behaviour is used in the method call to obtain the next behaviour to execute.

The agents running on mobile devices uses the Android application to specify what behaviours to execute in the Instance IB. Figure 13 shows the how these agents receive behaviours to execute

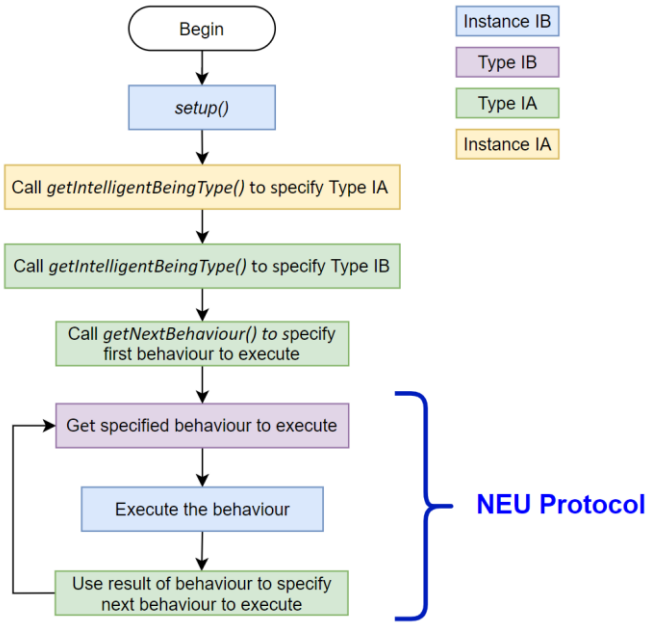


Figure 12: Flow of diagram NEU protocol.

The Instance IB obtains its corresponding Type IB in the same way as the agents running on a PC (as presented in Figure 12). The Instance IB then sends a broadcast to change the application frame according to the new agent type. The user then interacts with the application, which in turn calls the *executeBehaviour()* method in the O2A interface to specify which behaviour the Instance IB must obtain from the Type IB to execute. The behaviour which the Instance IB obtains and executes depends on the arguments received by the *executeBehaviour()* method.

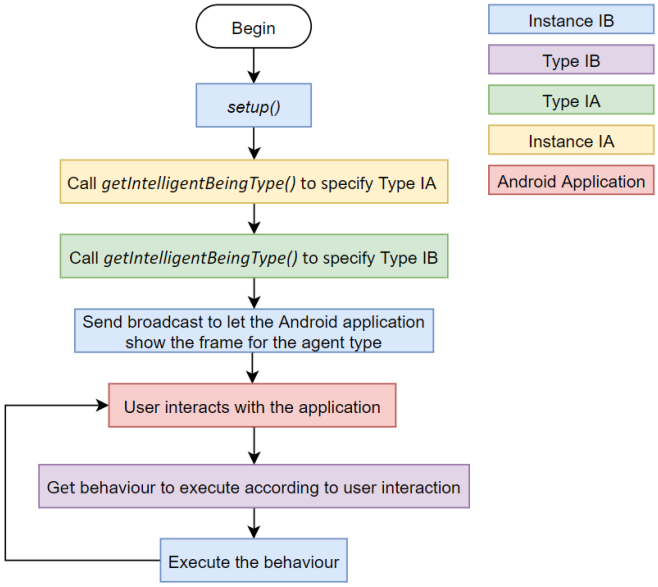


Figure 13: Flow diagram of Android application Instance IB.

5.2.2 Activity Holon Implementation

5.2.2.1 Activity Type Intelligent Being

The ATIB components only contain a single *getFSMBehaviour()* method that takes no arguments. When this method is called, it returns a Finite-State Machine (FSM) behaviour containing all the activity-specific behaviours. The behaviours of the production order are presented in Table 1.

The resource selection activity only contains an *assignResourceCNP* behaviour in its FSM behaviour. This behaviour executes the CNP, as discussed in section 2.1.3.6, to negotiate the selection of resources to assign to a production order.

Table 1: ATIB behaviours for a production order activity

Behaviour	Description
initiateNewSelectionActivity	*OneShotBehaviour* that initiates a new resource selection activity to select resources to assign to a production order
terminateProductionOrderActivityInstance	*OneShotBehaviour* that terminates the production order AIIB agent
handleRequestMessages	*SSResponderDispatcher* behaviour that creates a *SSIteratedAchieveREResponder* behaviour in a dedicated thread to respond to the received message

5.2.2.2 Activity Type Intelligent Agent

The ATIA components only contain the methods to make activity-specific decisions. The methods of the production order and resource selection activities are respectively presented in Tables 2 and 3.

Table 2: ATIA methods for a production order activity

Method	Description
String getIntelligentBeingType()	Return the ATIB containing the activity-specific execution functionality
DataStore buildDataStore (Object[] arguments)	Return the agent's DataStore after storing the received arguments in the DataStore
Int getNextBehaviour (String currentbehaviour, String currentSelAct)	Uses the currently executing behaviour and the current resource selection activity to return an Int used to select the next behaviour to execute
String getNextActivityInstanceType (String currentSelectionActivity)	Uses the current resource selection activity to return the String of the next resource selection activity to initiate
Boolean startNewActivityInstance (String currentSelectionActivity)	Uses the current selection activity to return a Boolean that specifies if a new resource selection activity should be initiated or not
Int getResponderBehaviour (String msgOntology)	Uses the ontology of the received message to return an Int that is used to select the behaviour to respond to the received message
String getHashMapKey (String selectionActivityType)	Uses the resource selection activity type to return the String used as the key in the HashMap where the information of the activity is stored
HashMap<String, String> getPalletProfile (String boxtype)	Uses the type of carton box packing material to return the HashMap containing the packing information for the type of carton box used to determine the production order target values
String getResourceSelectionActivity (String newResourceType)	Return a String of the new resource selection activity to initiate according to the String of the new resource type received

Table 3: ATIA methods for a resource selection activity

Methods	Description
String getIntelligentBeingType()	Return the ATIB containing the activity-specific execution functionality
DataStore buildDataStore (Object[] arguments)	Return the agent's DataStore after storing the received arguments in the DataStore
String getDecisionMakerResourceType()	Return the type of decision-making resource as a String to make the resource selection decisions

5.2.2.3 Activity Instance Intelligent Agent

The AIIA components direct the AIIB to the correct ATIA for activity-specific decision making. The AIIA component of both the production order and resource selection activities contain the same methods. These methods are presented in Table 4, with a short description on each method.

Table 4: AIIA methods for a production order and resource selection activity

Methods	Description
String getIntelligentBeingType (String agentType)	Return the ATIB obtained from the ATIA according to the agent type String received
DataStore buildDataStore (String agentType , Object[] arguments)	Return the agent's DataStore obtained from the ATIA according to the agent type String received

5.2.2.4 Activity Instance Intelligent Being

5.2.2.4.1 Production Order Activity - Resource Assignment

To assign new resources, the production order AIIB executes the flow diagram presented in Figure 14. The AIIB calls the *getNextBehaviour()* method (Table 2) which specifies the behaviour to execute. The AIIB then obtains the *initiateNewSelectionActivity* behaviour (Table 1) from the ATIB and executes the behaviour. The behaviour initiates the resource selection activity specified by the *getNextActivityInstanceType()* method (Table 2). The result of the execution of the behaviour is then used to specify the next behaviour to execute in the AIIB.

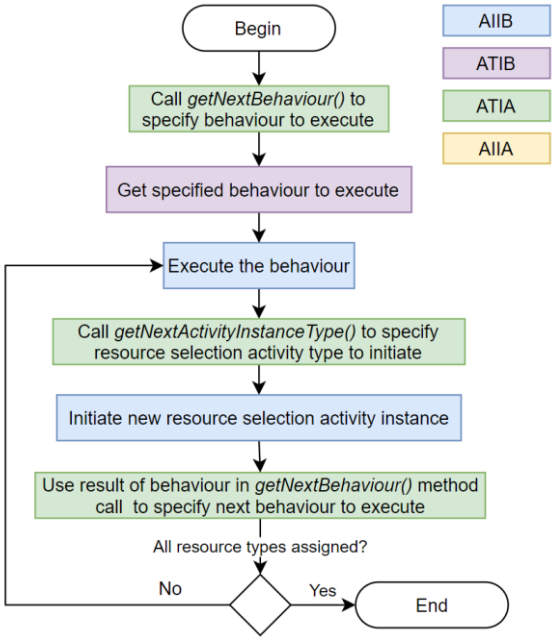


Figure 14: Flow diagram of a production order AIIB assigning new resources.

5.2.2.4.2 Production Order Activity - Receiving Messages

To receive messages, the production order AIIB executes the flow diagram presented in Figure 15. The production order AIIB receives messages using the *handleRequestMessages* behaviour (Table 1). This behaviour is executed when the AIIB agent is initiated and continues to run in the background until a message is received. When a message is received, the AIIB calls the *getResponderBehaviour()* method (Table 2) to specify the *SSIteratedAchieveREResponder* behaviour which the *handleRequestMessages* must execute in response to the message received. The type of messages the production order AIIB can receive and their responses are presented in Table 5.

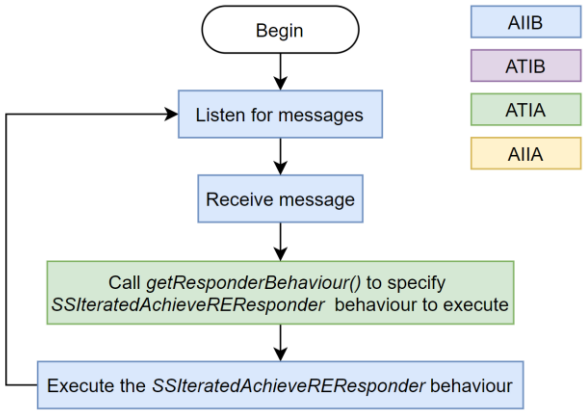


Figure 15: Flow diagram of a production order AIIB receiving a message.

Table 5: SSIteratedAchieveREResponder behaviours to respond to messages

Received Message	SSIteratedAchieveREResponder Behaviour Description	Reply
Request message to update assigned resources	Adds the list of resources received to the HashMap containing the list of assigned resources	Inform message to acknowledge assigned resources updated
Request message to obtain production order information	Determine the completion progress according to the resource's progress and target value	Inform message containing the percentage complete
Request message to cancel production order	Sends a request to all assigned resources to remove the production order form their schedule before terminating	Inform message to acknowledge the cancellation
Request message to obtain list of assigned resources	Create an XML String containing all assigned resources	Inform message containing assigned resources XML String
Request message to remove an assigned resource	Send request to the resource to remove the production order from its schedule before removing the resource from the assigned resources list	Inform message to acknowledge the removal of the resource
Request message to assign a new resource	Create new resource selection activity to assign resources	Inform message to acknowledge resource assignment
Request message to update the progress of a task	Update the progress of a task	Inform message to acknowledge task progress updated

5.2.2.4.3 Resource Selection Activity - Selecting Resources

To select resources to assign to a production order, the resource selection AIIB executes the flow diagram presented in Figure 16. The resource selection activity AIIB executes the *assignResourceCNP* behaviour (Table 2). The AIIB obtains the resources available to perform the required service from the DF, before sending each of these resources a CFP. On receipt of the proposals from the resources, the AIIB calls the *getDecisionMakerResourceType()* method to obtain the decision maker type. The decision maker type is used to obtain the decision-making resource from the DF, before sending the proposals to the decision-making resource. On receipt of the selected resources from the decision maker, the AIIB sends a message to all the resources that sent proposals, to inform them if their proposals were accepted or rejected. The AIIB then also sends the list of selected resources to the production order activity that initiated the resource selection activity to assign the resources to the production order.

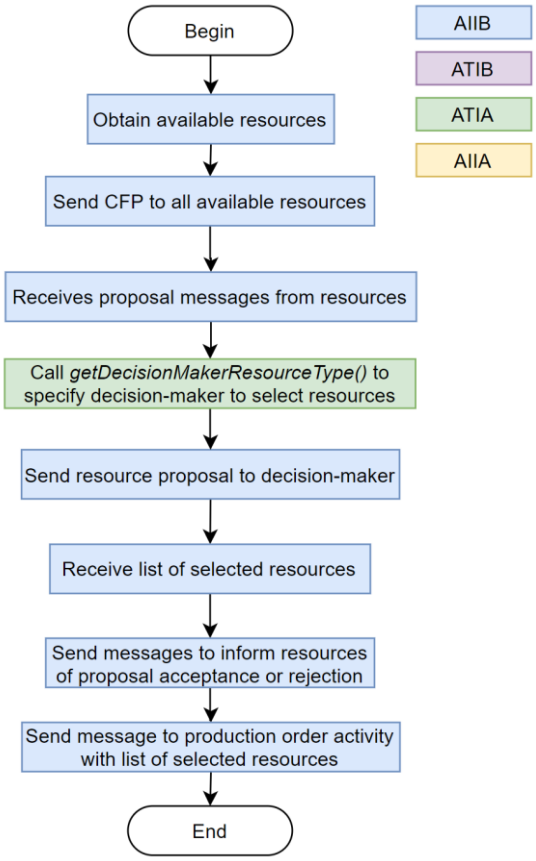


Figure 16: Flow diagram of a resource selection AIIB selecting resources to assign to a production order.

5.2.3 Resource Holon Implementation

5.2.3.1 Resource Type Intelligent Being

The RTIB components contain different methods according to the functionality requirement of the resource. The methods of the different resource types are discussed in sections 5.2.3.1.1 – 5.2.3.1.6.

5.2.3.1.1 Resources Running on a PC

The vineyard and packhouse resources are asset and facility resources as described in section 3.1. The vineyard and packhouse resources run on a PC as described in section 5.1.1. Although the vineyard and packhouse resources differ in execution, the RTIB of both resources have the same implementation. The ATIB components only contain a single *getFSMBehaviour()* method that takes no arguments. When this method is called, it returns an FSM behaviour containing all the activity-specific behaviours. The behaviours in the FSM behaviour are presented in Table 6, with a short description of each.

Table 6: RTIB methods for resources running on a PC

Behaviours	Description
initiateHandleResourceAssignmentRequests	*OneShotBehaviour* that initiates a ContractNetResponder behaviour to perform the CNP discussed in section 2.1.3.6
handleRequestMessages	*SSResponderDispatcher* behaviour that creates a *SSIteratedAchieveREResponder* behaviour in a dedicated thread to respond to the received message
terminateActivityInstance	*OneShotBehaviour* that terminates the RIIB agent

5.2.3.1.2 Resources Running on a Mobile Devices

The packing material storage facility is an asset and facility resource as described in section 3.1 and runs on a mobile device as described in section 5.1.1. The harvesting teams, quality control stations, packing stations and vehicles are human task performers as described in section 3.2 and also run on a mobile device as described in section 5.1.1. Although the purpose and execution of these resources differ, their RTIB components have the same implementation and contains the methods presented in Table 7, with short description of each method. Each of these methods return a behaviour which is added to the agent to be executed. The *handleRequestMessages()* and *handleResourceAssignments()* methods are called when the RIIB agent is created to initiate the behaviours that enables the agent to receive messages and handle resource assignments.

Table 7: RTIB methods for resources running on a mobile device

Methods	Description
ContractNetResponder handleResourceAssignments (Object arguments)	Return a ContractNetResponder behaviour to perform the CNP
SSResponderDispatcher handleRequestMessages (Object arguments)	Return a *SSResponderDispatcher* behaviour that creates a *SSIteratedAchieveREResponder* behaviour in a new thread to respond to the received message
OneShotBehaviour updateAgentDataStore (Object arguments)	Return a OneShotBehaviour behaviour to update the information in the agent's DataStore
AchieveREInitiator updateResourceTarget (Object arguments)	Return an AchieveREInitiator behaviour to update the progress made towards a production order and request the new target value
AchieveREInitiator requestProductionOrderRemoval (Object arguments)	Return an AchieveREInitiator behaviour to request the resource's removal from a production order

5.2.3.1.3 Production, Farm and Packhouse Managers

The production, farm and packhouse managers are also human task performers as described in section 3.2 and runs on a mobile device like the resources in section 5.2.3.1.2. However, the tasks which these resources perform are different. Therefore, the RTIB component is different and contains the methods presented in Tables 8-10, with a short description of each. The *receiveRequestMessages()* method is executed when the RIIB agent is created to enable the agent to receive messages.

Table 8: RTIB methods for a production manager resource

Methods	Description
SSResponderDispatcher receiveRequestMessages (Object arguments)	Return a *SSResponderDispatcher* behaviour that creates a *SSIteratedAchieveREResponder* behaviour in a new thread to respond to the received message
AchieveREInitiator sendSelectedResources (Object arguments)	Return an *AchieveREInitiator* to send the selected resources to the resource selection activity
AchieveREInitiator createProductionOrder (Object arguments)	Return an *AchieveREInitiator* to request a new production order activity be created
AchieveREInitiator cancelProductionOrder (Object arguments)	Return an *AchieveREInitiator* to cancel a production order by requesting its termination
TickerBehaviour updateActiveProductionOrders (Object arguments)	Return a TickerBehaviour that periodically requests the active production orders from the DF
AchieveREInitiator retrieveAssignedResources (Object arguments)	Return an *AchieveREInitiator* to request the assigned resources from a production order
AchieveREInitiator retrieveResourceInformation (Object arguments)	Return an *AchieveREInitiator* to request the information of a resource
AchieveREInitiator retrievePreviousProductionOrders (Object arguments)	Return an *AchieveREInitiator* to request the previous production orders from the component manager
AchieveREInitiator retrievePreviousProductionOrderInformation (Object arguments)	Return an *AchieveREInitiator* to request the information on a previous production order from the component manager
AchieveREInitiator removeProductionOrderResource (Object arguments)	Return an *AchieveREInitiator* to request that a resource be removed from a production order
AchieveREInitiator addNewProductionResource (Object arguments)	Return an *AchieveREInitiator* to request that a new resource be added to a production order

Table 9: RTIB methods for a farm manager resource

Methods	Description
AchieveREInitiator getFarmResources (Object arguments)	Return an *AchieveREInitiator* to obtain the resources belonging to the farm
AchieveREInitiator createFarmResource (Object arguments)	Return an *AchieveREInitiator* to request that the component manager create a new farm resource
SSResponderDispatcher receiveRequestMessages (Object arguments)	Return a *SSResponderDispatcher* behaviour that creates a *SSIteratedAchieveREResponder* behaviour in a new thread to respond to the received message
AchieveREInitiator sendSelectedResources (Object arguments)	Return an *AchieveREInitiator* to send the selected resources to the resource selection activity
AchieveREInitiator retrieveResourceInformation (Object arguments)	Return an *AchieveREInitiator* to request the information of a resource
AchieveREInitiator updateFarmManagerResource (Object arguments)	Return an *AchieveREInitiator* to request that a resource update its information according to the content of the request
AchieveREInitiator terminateResource (Object arguments)	Return an *AchieveREInitiator* to request that a farm resource terminate

Table 10: RTIB methods for a packhouse manager resource

Methods	Description
AchieveREInitiator updatePackhouseData (Object arguments)	Return an *AchieveREInitiator* to request that a packhouse update its information according to the content of the request
AchieveREInitiator requestPackhouseData (Object arguments)	Return an *AchieveREInitiator* to request the latest information from a packhouse resource
SSResponderDispatcher receiveRequestMessages (Object arguments)	Return a *SSResponderDispatcher* behaviour that creates a *SSIteratedAchieveREResponder* behaviour in a dedicated thread to respond to the received message
AchieveREInitiator requestProductionOrderRemoval (Object arguments)	Return an *AchieveREInitiator* to request that a packhouse resource remove itself from a production order

5.2.3.1.4 Component Manager

The component manager is an additional resource added to create components in the system as described in section 4.1. Since the production and farm managers cannot access files on a PC, they cannot create agents in the main container running on a PC. Therefore, the component manager runs on the PC hosting the main container to create activity and resource agents according to received request messages. The RTIB component only contains a single *getFSMBehaviour()* method that takes no arguments. When this method is called, it returns an FSM behaviour containing the activity-specific behaviours. The FSM behaviour only contains a *handleRequestMessages* behaviour that is a *SSResponderDispatcher* behaviour that creates a *SSIteratedAchieveREResponder* behaviour in a dedicated thread to respond to the received message. Although the FSM behaviour only contains a single behaviour, using the FSM behaviour makes provision for the addition of behaviours.

5.2.3.2 Resource Type Intelligent Agent

The RTIA components only contain the methods to make resource-specific decisions. The RTIA component of all resources running on a mobile device contain the same methods – presented in Table 11. The vineyard and packhouse resources run on a PC, therefore they require an additional method added to the bottom of Table 11.

Table 11: RTIA methods of the resources

Methods	Description
String getIntelligentBeingType()	Return the ATIB containing the activity-specific execution functionality
DataStore buildDataStore (DataStore dataStore, String resourceType, Object[] arguments, Vector<String> behaviourInformation)	Return the agent's DataStore after storing the received variables in the DataStore
Int getNextResponderBehaviour (ACLMessage msgRequest)	Uses the ontology and performative of the received message to return an Int that is used to select the behaviour to respond to the received message
Additional behaviour in the RTIA of the vineyard and packhouse resources:	
Int getNextBehaviour (String currentbehaviour)	Uses the behaviour currently executing the current resource selection activity to return an Int that is used to select the next behaviour to execute

5.2.3.3 Resource Instance Intelligent Agent

The RIIA components directs the RIIB to the correct RTIA for resource-specific decision making. The RIIA component of all the resources contain the same methods. These methods are presented in Table 12 with a short description of each.

Table 12: RIIA methods for a production order and resource selection activity

Methods	Description
String getIntelligentBeingType (String agentType)	Return the RTIB obtained from the RTIA according to the agent type String received
DataStore buildDataStore (String agentType , Object[] arguments)	Return the agent's DataStore obtained from the ATIA according to the agent type String received

5.2.3.4 Resource Instance Intelligent Being

5.2.3.4.1 Resource Assignment to Production Order

For resources to be assigned to a production order, the resources execute the flow diagram presented in Figure 17. The RIIB executes the *ContractNetResponder* behaviour as described in Tables 6 and 7. The RIIB receives the CFP from the resource selection activity. The RIIB responds by sending a proposal containing the information of the resource. Each resource then receives either an *accept-proposal* or a *reject-proposal* message if they were selected or not before responding with an *inform* message to acknowledge the acceptance or rejection.

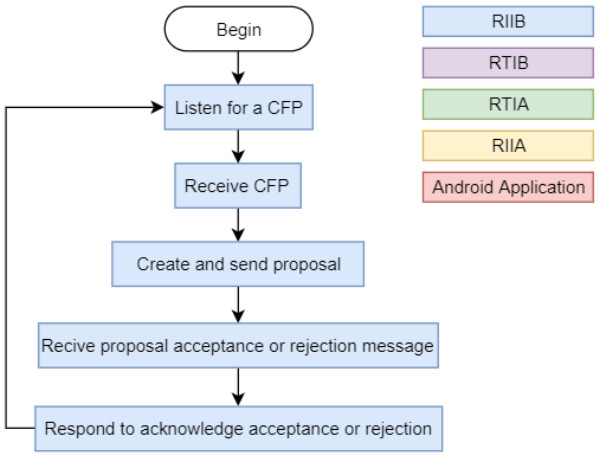


Figure 17: Flow diagram of a resource handling production order assignments.

5.2.3.4.2 Resource Updating Production Order Task Progress

To update a production order with a resource's latest progress, the RIIB executes the flow diagram presented in Figure 18. The user interacts with the Android application to update the progress of a resource. The RIIB receives the updated progress from the application through the O2A interface. The RIIB then gets the *AchieveREInitiator* behaviour (Table 7) to execute according to the update information request received from the Android application. The RIIB then executes the behaviour to send a message to update the production order task progress.

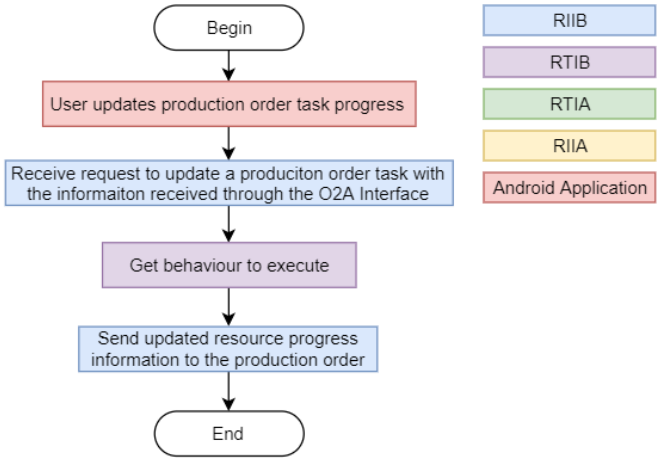


Figure 18: Flow diagram of a resource updating a production order task.

5.2.3.4.3 Decision Maker Selecting Resources

To select resources to assign to a production order, the decision maker RIIB executes the flow diagram presented in Figure 19. The RIIB receives a message to request the selection of resources for a production order with the *SSResponderDispatcher* behaviour obtained from the *handleRequestMessages()* method (Table 8 and 9). The RIIB then gets the *SSIteratedAchieveREResponder* behaviour to execute and passes the proposals to the Android application by calling the *getNextResponderBehaviour()* method (Table 11).

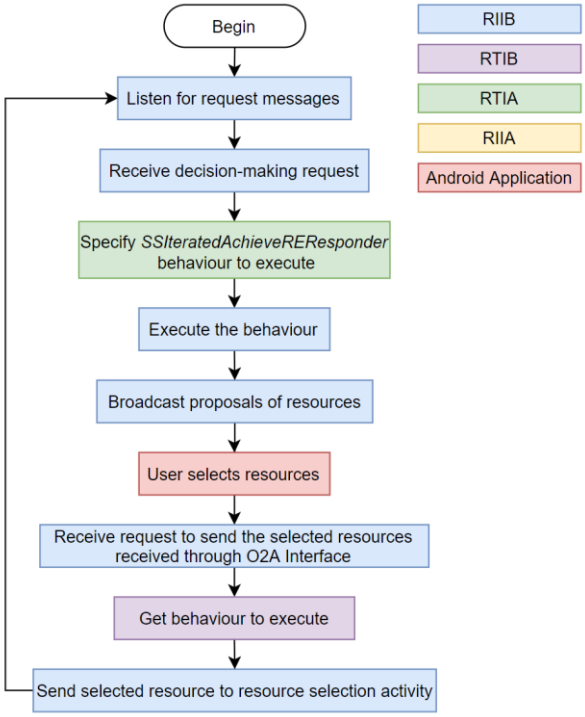


Figure 19: Flow diagram of a decision maker resource selecting resources.

The user interacts with the Android application to select resources to assign to the production order. The RIIB then receives the list of selected resource form the Android application through the O2A interface. The RIIB then executes the behaviour to send the list of assigned resources accordingly.

5.2.3.4.4 Resource with Android Application - Sending Messages

To send messages, the RIIB with an Android application executes the flow diagram presented in Figure 20. The user interacts with the Android application to specify the message and information to send. The RIIB gets the *AchieveREInitiator* behaviour to execute accordingly. The messages which the respective resources can send with their *AchieveREInitiator* behaviours are presented in Tables 8-10.

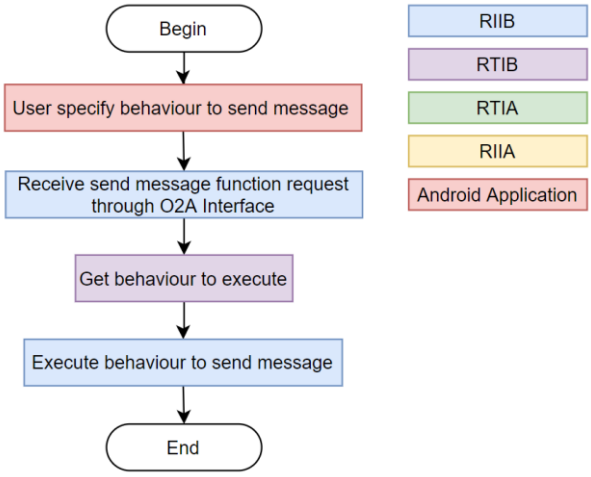


Figure 20: Flow diagram of an Android application sending a message.

5.2.3.4.5 Resources Receiving Messages

To receive messages, the RIIB executes the flow diagram presented in Figure 21. The respective RIIBs receive messages with their *SSResponderDispatcher* behaviour (Tables 6-10). The resulting *SSIteratedAchieveREResponder* behaviour to execute is specified by calling the *getNextResponderBehaviour()* method (Table 11). The respective *SSIteratedAchieveREResponder* behaviours are presented in Table 13.

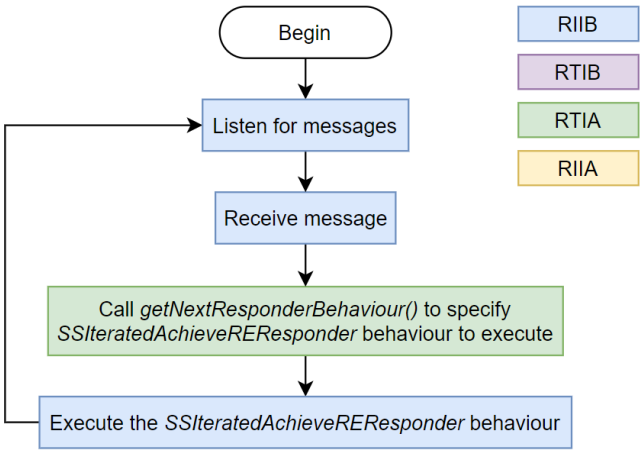


Figure 21: Flow diagram of a resource receiving a message.

Table 13: SSIteratedAchieveREResponder behaviours to respond to messages

Received Message	SSIteratedAchieveREResponder Behaviour Description	Reply
Request resource termination	PC resource send acknowledgement before terminating the resource agent	Inform message to acknowledge resource termination
Request production order removal	Resource remove the production order from the resource's schedule	Inform message to acknowledge production order removal
Request resource information update	Resource update the information of a resource before sending response	Inform message to acknowledge resource information update
Request resource information	Resource create a message containing the resource's information in XML String	Inform message containing resource information XML String
Request agent creation	Component manager create activity or resource agent according to request content	Inform message to acknowledge resource creation
Request previous production orders	Component manager read information from text file and send previous production orders and their information	Inform message containing the previous production orders and their information

5.3 Holon Interactions

The holons are autonomous and cooperative entities. Holons interact and cooperate to perform the system tasks. This section will discuss the interactions among holons, with the focus on the main tasks the system needs perform.

5.3.1 Initiate Production Order

The production manager has the functionality to initiate a new production order, as described in section 5.2.2.3.3. As shown in Figure 22, the production manager agent sends a request to the component manager agent to create a production order agent. The component manager then informs the production manager agent if it was successful in creating the production order agent or not.

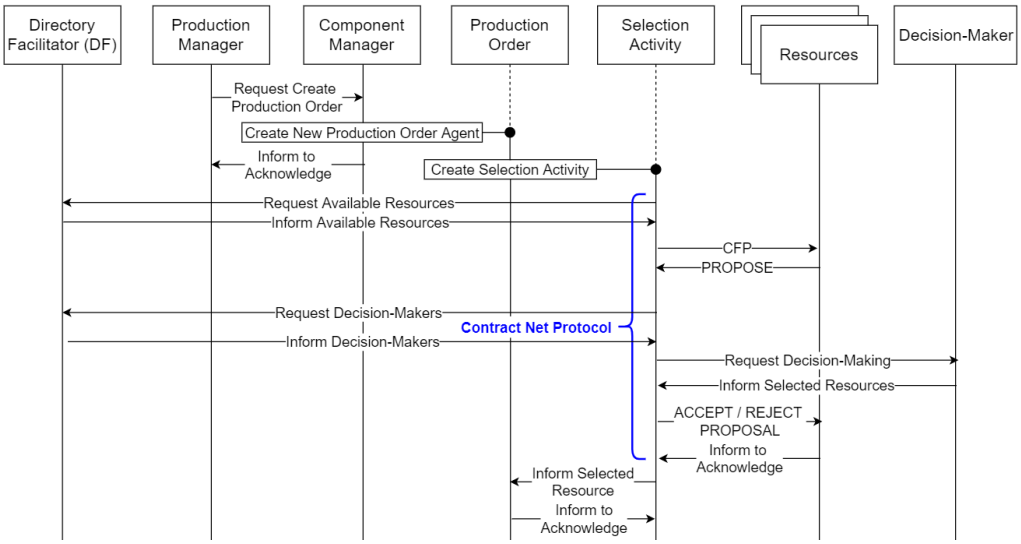


Figure 22: Production order activity initiation and resource assignment sequence diagram

When a production order agent is created, it initiates a resource selection activity agent. The resource selection activity agent acquires the resources available for a specific service from the DF. The selection activity then sends a CFP to each of these resources. The resources reply with proposals upon receipt of the CFP. Once the selection activity agent received proposals from all the resources, it acquires the available decision makers from the DF, according to the resource type to assign. The decision maker can either be a production or farm manager.

The decision maker receives a request from the selection activity to select resources for a production order. The decision maker agent then sends the request to the Android application where the decision maker can select resources according to their proposals. The selected resources are then sent to the selection activity agent by the decision maker agent. Once the selection activity receives the selected resources, the selection activity sends *accept-proposals* to the resources that have been selected and *reject-proposals* to the resources not selected. Once all the resources responded with an *inform* message to acknowledge the receipt of either an *accept-proposal* or *reject proposal*, the selection activity sends the selected resources to the production order agent before terminating.

The production order agent then updates its list of assigned resources according to the selected resources received. If the production order agent requires more of a different type of resource, a new selection activity agent is created to repeat the above-mentioned resource selection sequence of events. If all the resources are assigned, the production order agent will listen for incoming messages.

5.3.2 Manage Production Order Resources

Figure 23 shows how a production manager can add or remove a production order agent. In both cases the production manager agent sends a *request* message to the production order agent. To add a resource, the production order agent will initiate a selection activity agent, discussed in section 5.3.1, before informing the production manager agent of the result. If the request is to remove a resource, the production order agent will remove the resource from its list of assigned resources before sending an *inform* message to acknowledge the removal of the resource.

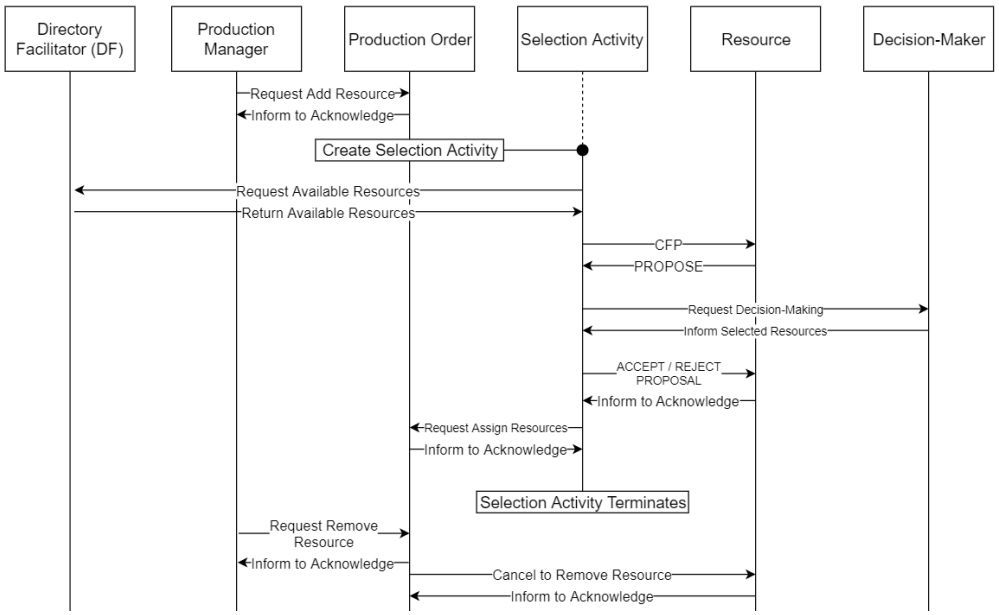


Figure 23: Sequence diagram of production manager adding and removing a resource from a production order.

5.3.3 Terminate Production Order

Figure 24 shows how a production manager cancels a production order. The production manager agent sends a termination request to the production order agent. The production order agent then sends an inform to the production manager agent to acknowledge the request, before sending a *request* message to all the assigned resources to remove the production order from their schedule. The resources respond with an inform to acknowledge. The production order agent then terminates upon receipt of all the *inform* messages from the resources.

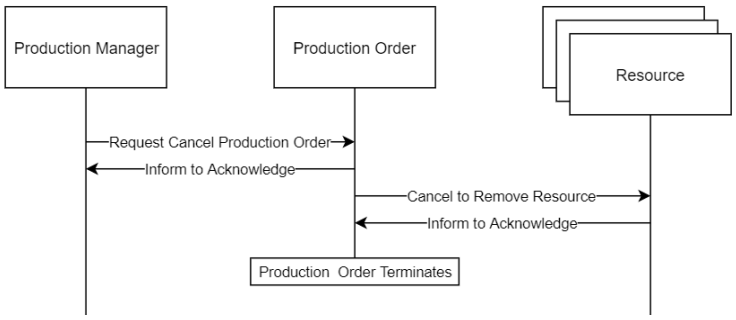


Figure 24: Sequence diagram of a production manager terminating a production order.

5.3.4 Monitor Production Order

Figure 25 shows how a production manager monitors a production order. The production manager agent periodically requests the latest information about the active production order agents. First, the active production order agents are obtained from the DF, before sending a *request* message to each active production order agent to request their latest progress information. The production order agents respond with an inform containing its latest progress information.

The progress of a production order is updated each time a resource updates its progress towards a production order task by sending its latest production order task progress to the production order agent. The production order agent responds by sending the updated target value to the resource. When target value for a production order task is reached, the task is complete. The production order agent then removes the resources from its list of assigned resources and sends a *cancel* message to the resources to remove the production order from their schedule.

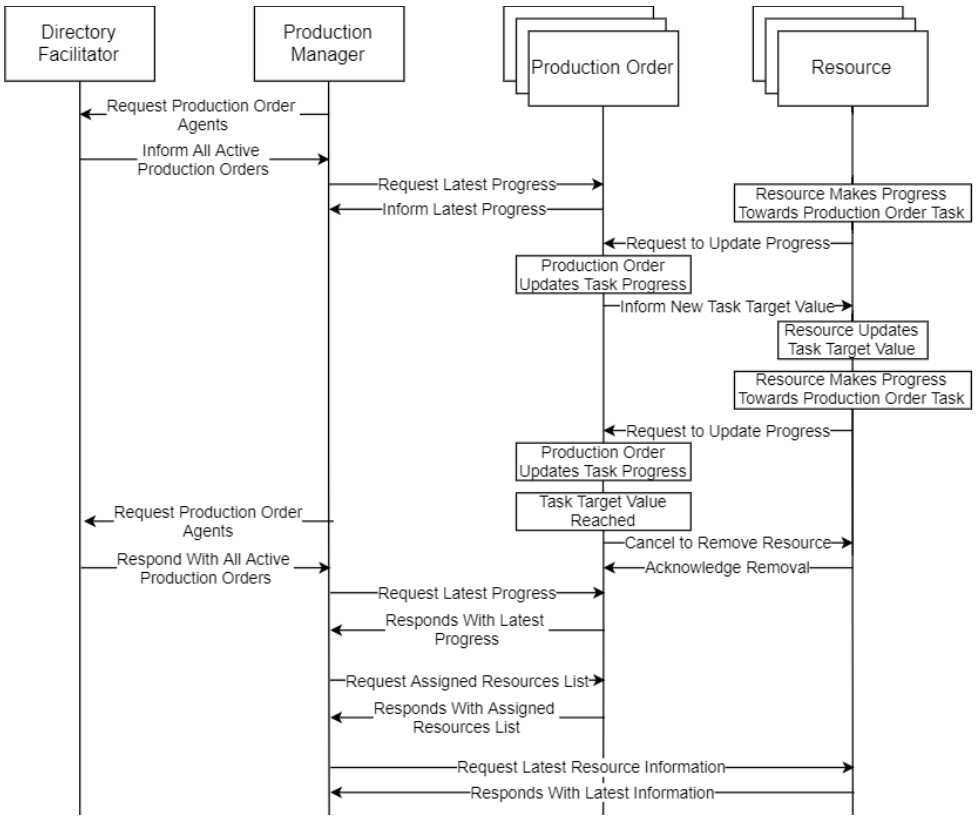


Figure 25: Sequence diagram of a production manager monitoring a production order.

The production manager can also examine a production order. The production manager agent can request the assigned resources of an active production order agent. The production order agent then responds with an inform containing the assigned resources. The production manager can then examine an assigned resource by request the resource's information. The resource responds with an inform containing its latest information.

5.3.5 Managing Farm Resources

The farm manager can add or remove a packhouse or vineyard resource on its farm. The other resources on the farm become active when a user logs in on the Android application and terminates when a user logs out. The farm manager can also monitor each resource on the farm. Figure 26 shows the sequence of message exchanges for a farm manager to create, terminate and update a farm resource.

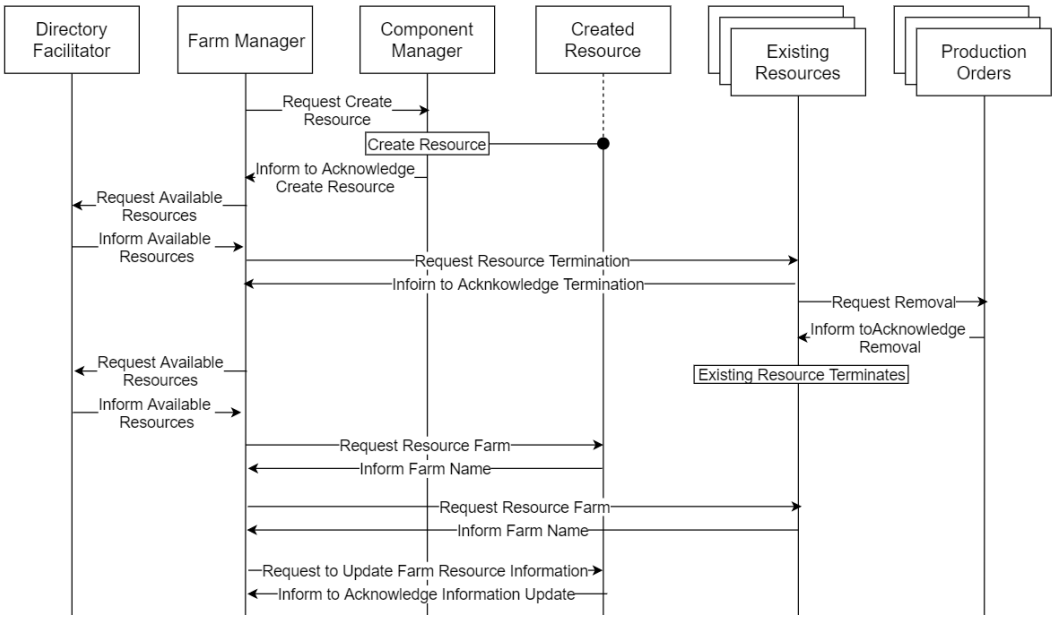


Figure 26: Sequence diagram of a farm manager creating and terminating a resource and acquiring and updating a resource.

The farm manager can create a packhouse or vineyard resource by sending the specifications in a *request* message to the component manager agent. The component manager agent then informs the farm manager agent if it was successful in creating the resource agent. The farm manager can also terminate a packhouse or vineyard resource by sending a termination *request* message to a resource. The resource responds with an *inform* message and sends a *request* message to all the production order agents in its schedule to request its removal, before terminating.

The farm manager can inspect the resources of the farm he manages by first acquiring the active resources from the DF. The farm manager agent then requests the farm each of these resources belong to, since the DF only contains the service a resource provides. The resources respond with an *inform* message containing their farm. On receipt of each message, the farm manager RTIA determines if the resource belongs to the farm or not. The farm manager agent can then send a *request* message to a resource to obtain its information. The resource responds with an *inform* message containing its latest information. The farm manager can then update the information of a packhouse or vineyard resource. The farm manager agent sends a *request* message containing the updated information. The resource updates its information according to received message and responds with an inform to acknowledge the update

5.3.6 Previous Production Order Information

Figure 27 shows how a production manager agent can obtain the previous production orders and their information by sending a *request* message to the component manager agent. The component manager agent obtains the list previous production orders before responding with an *inform* message containing the previous production orders. The production manager agent can request the information on a specific previous production order from the component manager agent. The component manager agent obtains the information before responding with an *inform* message containing the information.

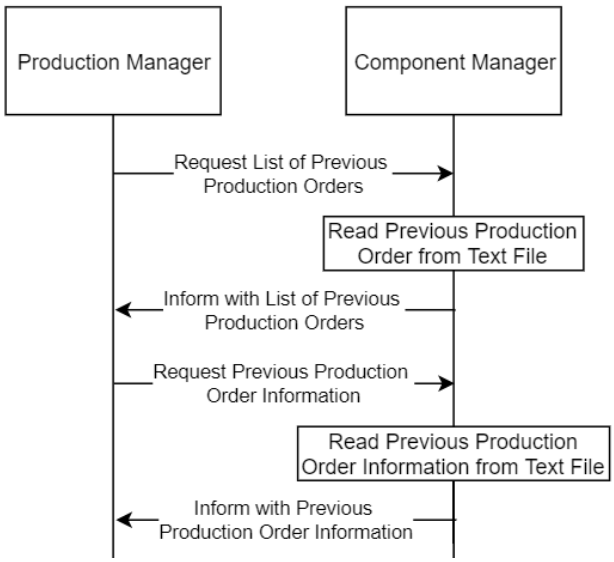


Figure 27: Sequence diagram of a production manager acquiring the previous production orders and their information.

6 Evaluation

This chapter discusses the evaluation of the DS. The evaluation entails the verification the system's functionality by comparing the DS to the CS. The criteria used to evaluate both the DS and CS are discussed, followed by the experiments on which the criteria are used to determine each system's performance and, finally, the results are presented and discussed.

6.1 Evaluation Criteria

The evaluation criteria used to determine each system's performance are discussed in the following sections. The evaluation criteria consider how the system communicates information, how the system manages information and what support the system provides for making decisions.

6.1.1 Communication

The communication of information will be evaluated in terms of accuracy and reliability. The accuracy of the communication refers to the certainty that the end receiver will receive the message as intended by the sender. The reliability of the communication refers to the certainty that the system will consistently deliver the messages to the intended receiver and that action will be taken accordingly.

The accuracy of communication will be evaluated by:

- Counting the number of times a message has to be exchanged before it reaches the intended receiver.
- Determining if the message can be altered before it reaches the intended receiver.
- Determining if the messages are structured according to a specific format.

Each time a message is exchanged there is a chance that the message is wrongly interpreted and altered by the receiver before it is forwarded to the next receiver. Therefore, the communication accuracy can be increased by reducing the number of message exchanges, inhibiting message alterations and consistently using a specific message format to prevent misinterpretations.

The reliability of communication will be evaluated by:

- Counting the number of message exchanges required before the message reaches the intended receiver.
- Determining if the receipt of the message is acknowledged.
- Determining if the sender knows the availability of the receiver to act when receiving a message.

With each message exchange the message can be received without being forwarded to the next receiver. Therefore, reducing the number of message exchanges increases the communication reliability. If the receiver responds with an acknowledgement, the sender is certain that the message was received. Knowing that the message is received increases the communication reliability. Knowing if a receiver is available to receive messages and act accordingly further increase the communication reliability.

6.1.2 Information Management

The information management will be evaluated by considering the accessibility and traceability of the system's information. In this context, accessibility refers to the dynamic, instance-specific information on active production orders and the system's ability to obtain and communicate this information to the relevant personnel. Traceability refers to the ability of the system to store and recall stored information on current and previous production orders.

The accessibility of information will be evaluated by:

- Examining how the system obtains instance-specific information.
- Counting the number of messages required to be sent in order to obtain the instance-specific information.
- Determining how close to real time does the information represent the PM.

A system that obtains the instance-specific information automatically, compared to a system where the information needs to be requested manually, will have better accessibility of information. Each message that must be sent in order to obtain the instance-specific information adds to the latency to obtain information. Therefore, the accessibility of the system can be increased by reducing the number of messages sent when information is requested. The accessibility of a system can further be increased by providing information as close to real time as possible.

The traceability of information will be evaluated by:

- Determining if information about the production orders is stored or not, and where it is stored.
- Determining if the stored information is structured in a specific format.
- Determining if the stored information can be accessed across multiple devices.
- Determining who can access the stored information.

Storing information about production orders improves the traceability of a system by allowing the system to recall information that can be used to study the execution of previous production orders. This information can be used to improve future production orders. Storing the information in a structured format eases the retrieval of specific information on a specific production order, increasing the traceability of the system. Having a system that is able is provide the stored information across multiple devices to anyone who may require the information further increases the traceability.

6.1.3 Decision Support

The decision support offered by the system will be evaluated in terms of consistency and agility. Consistency refers to the system's ability to ensure decisions are made consistently by the same type of decision makers, with the same information available. The agility refers to the system's ability to support changes to previous decisions, e.g. decisions related to resource assignments.

The consistency of the decision support will be evaluated by:

- Determining how the system manages user permissions.
- Determining if, and how, a structured decision-making process is supported.
- Determining if the system provides the same type of information for the current decision, and next decisions of the same type.

To ensure the decisions are consistent across multiple production orders, the system should only allow production order decisions to be made by production managers and farm decisions to be made by farm managers. To achieve consistent decision making, a structured decision-making process should be supported. The same type of information should be provided for the same type of decisions to ensure informed decisions are consistently made.

The agility of the decision support will be evaluated by:

- Determining if, and how, the system provides information to support decisions to make changes to production orders.
- Determining how the decisions are conveyed.

Agile decision making relies on quick access to information to ensure decisions are made as quickly and informed as possible. An agile system will also convey the result of a decision to the relevant personnel as quickly as possible to ensure the changes are carried out correctly and as quickly as possible.

6.2 Experiment Design

The experiments to determine the functionality of the system are discussed in the following sections. The experiments consist of initiating a production order, monitoring an existing production order and the progress made by the assigned resources, making changes to an existing production order and cancelling an existing production order. The set of resources and the production specifications to be used in each experiment are presented in Table 14.

Table 14: Predetermined resource sets for experiments.

Experiment: Resources to assign for production order initiation								
Resources to Assign	Vineyard	Packhouse	Packing Material	Harvesting Team	Quality Control Station	Packing Station	Grape Transportation Vehicle	Packing Material Transportation
Resource Quantity / Specification	2	1	Carton Boxes Punnets Labels	1	1	1	1	1
Experiment: Initiate production order with limited available resources								
Resources to Assign	Vineyard	Packhouse	Packing Material	Harvesting Team	Quality Control Station	Packing Station	Grape Transportation Vehicle	Packing Material Transportation
Resource Quantity	1	1			1		1	1
Experiment: Resources assigned to production order to monitor								
Resources to Assign	Vineyard	Packhouse	Packing Material	Harvesting Team	Quality Control Station	Packing Station	Grape Transportation Vehicle	Packing Material Transportation
Resource Quantity / Specification	2	1	Carton Boxes Punnets Labels	1	1	1	1	1
Experiment: Changes to production order								
Resources to Assign	Vineyard	Packhouse	Packing Material	Harvesting Team	Quality Control Station	Packing Station	Grape Transportation Vehicle	Packing Material Transportation
Resource Changes	Remove a vineyard resource			Change the berry size of harvest grapes			Add a vehicle resource	Change the pickup location to a packhouse
Experiment: Resources to remove themselves from a production order								
Resources to Assign	Vineyard	Packhouse	Packing Material	Harvesting Team	Quality Control Station	Packing Station	Grape Transportation Vehicle	Packing Material Transportation
Resource Quantity				1		1	1	
Experiment: Assigned resources to production order to cancel								
Resources to Assign	Vineyard	Packhouse	Packing Material	Harvesting Team	Quality Control Station	Packing Station	Grape Transportation Vehicle	Packing Material Transportation
Resource Quantity / Specification	2	1	Carton Boxes Punnets Labels	1	1	1	1	1

6.2.1 Initiate Production Order

For a production order to exist, it must be initiated. This requires the assignment of all the resources necessary to complete the production order. The assignment of resources should be done by selecting at least one of each resource type. However, if there are resource types with no available resources, or no decision maker to assign a resource type, the system should still be able to initiation a production order with the available resource types and decision makers.

The DS will be compared to the CS when assigning the set of resources presented in Table 14 to initiate a production order. The system should determine the resources available, as well as gather information on each of the available resources to determine their suitability towards a production order. Once the resource assignments have been done, the assigned resources should be informed of their assignment to the production order. The resources should acknowledge their assignment to complete the production order initiation.

The DS will also be compared to the CS when assigning the limited set of resources presented in Table 14 to initiate a production order with limited resources. The systems should still be able to assign all the available resources from the available resource types. Again, the resources should be informed of their assignment to the production order. Both systems should be used to gather information on these resources to determine their suitability to the production order. Once all the resources acknowledged their assignment to the production order, the production order initiation is complete.

The DS will also be compared to the CS when initiating a production order with the limited decision makers. This entails that the systems should still be able to initiate a production order with only a production manager or farm manager, but not both. The system should only assign the resource types for which there is a decision maker available to select the resources to assign.

6.2.2 Monitor Production Order

The systems should provide decision makers with the ability to observe the progress made by the assigned resources, which allows them to proactively adjust production orders. The system must be able to gather the progress information from all the active production orders and present the information to the decision makers. The production orders also need to have the latest progress information of each assigned resource in order to provide the production order progress.

The systems will be compared on how they monitor the progress of an active production order. The set of resources assigned to the production order are provided in Table 14 and contains at least a single resource from each resource type. The set of resources also contain multiple resources from a single resource type to verify the functionality to monitor multiple resources.

The systems will be compared by gathering the latest production order progress information. The production orders should provide their completion progress on request, as well as the resources assigned to the production order. Each assigned resource should also be able to provide its latest progress information. The resources should also be able to update a production order whenever they make progress towards the production order's task.

6.2.3 Production Order Changes

The system should be able to adapt to production order changes in response to changes in the market or environment. This may require that the resources assigned to a production order be altered. Resources may be removed or new resources may be added. Assigned resources may also be replaced by removing an assigned resource and adding a new resource in its place. The production order should only have the resources actively contributing to the production order assigned to accurately represent the execution of the production order. Therefore, a resource must remove itself from production orders whenever the resource no longer contributes to the production order.

The DS will be compared to the CS with regards to making changes to an active production order with at least one of each resource type assigned. The changes will include removing and adding the set of resources provided in Table 14. The removal of resources includes informing the resources that they are no longer assigned to the production order to which the resources should react and respond to acknowledge their removal. Adding resources include obtaining the list of resources that are available to perform the required service and to gather information on these resources to select the best suited resource. The chosen resources then need to be informed of their assignment to the production order before they acknowledge their assignment.

Besides adding and removing resources, the system should also be able to make changes regarding the execution of tasks and resources removing themselves from a production order. The changes regarding the execution of tasks are provided in Table 14 and consists of changing the execution of a harvesting task. The changes regarding resources removing themselves are also provided in Table 14 to verify how the systems handles resources removing themselves.

6.2.4 Cancel Production Order

The systems should be able to cancel a production order due to changing environmental conditions or changing market demands. The cancellation of a production order entails releasing the resources from the production order and the resources removing the production order from their schedules.

The systems will be compared by cancelling an active production order containing at least one of each resource type. The cancellation requires that all the assigned resources be informed that the production order is cancelled. The resources must then remove the production order from their schedule before acknowledging the production order cancellation. The set of resources assigned to the production order to be cancelled are provide in Table 14.

6.3 Results and Discussion

The results of the experiments are provided in the following sections with a discussion on the performance of both the DS and CS. The results of initiating a production order, monitoring a production order, making changes to a production order and cancelling a production order are provided and discussed.

6.3.1 Initiate Production Order

The DS was used to initiate a production order containing at least one of each resource type as specified in Table 14. Appendix A.1 shows the sequence of all the messages exchanged between the activities and resources during the production order initiation experiments. The initiation functionality of the DS will be discussed using the sequence of message exchanges in Figure 28.

The light green lines represent the *request* send by the Production Manager to the Component Manager to create a production order activity, and the response sent by the Component Manager specifying if the production order was successfully created or not. The Component Manager first verifies if the requested production order does not yet exist (purple lines) by sending a *request* message to the DF to enquire if such a production order agent already exists. According to Figure 28, the TEST Production Order did not yet exist and was created. Had the requested production order already existed, the Component Manager would respond to inform the Production Manager that it failed to create the production order.

The blue lines represent the messages between the Component Manager and the JADE AMS to request the creation of the TEST Production Order agent and the response from the AMS to inform if the agent was successfully created.

The TEST production order first registers itself as a production order activity at the DF (yellow lines) before it creates a resource selection activity. The Vineyard Selection activity is created and it registers itself as a resource selection activity in the DF (red lines). This enables the system to obtain all the activities coordinating the same type of service. The Vineyard Selection activity then obtains all the available vineyard resources (black lines) from the DF.

Figure 28 only show the negotiations between the Vineyard Selection activity and the “Eilandia Flame Blok 1” vineyard resource (pink lines). The Vineyard Selection activity sends a CFP to all the available resources. The resources respond with their *proposal* messages containing their information that is used to selected resources to assign to the production order. Appendix B.1 shows the information sent from the vineyard resource as its proposal in XML format.

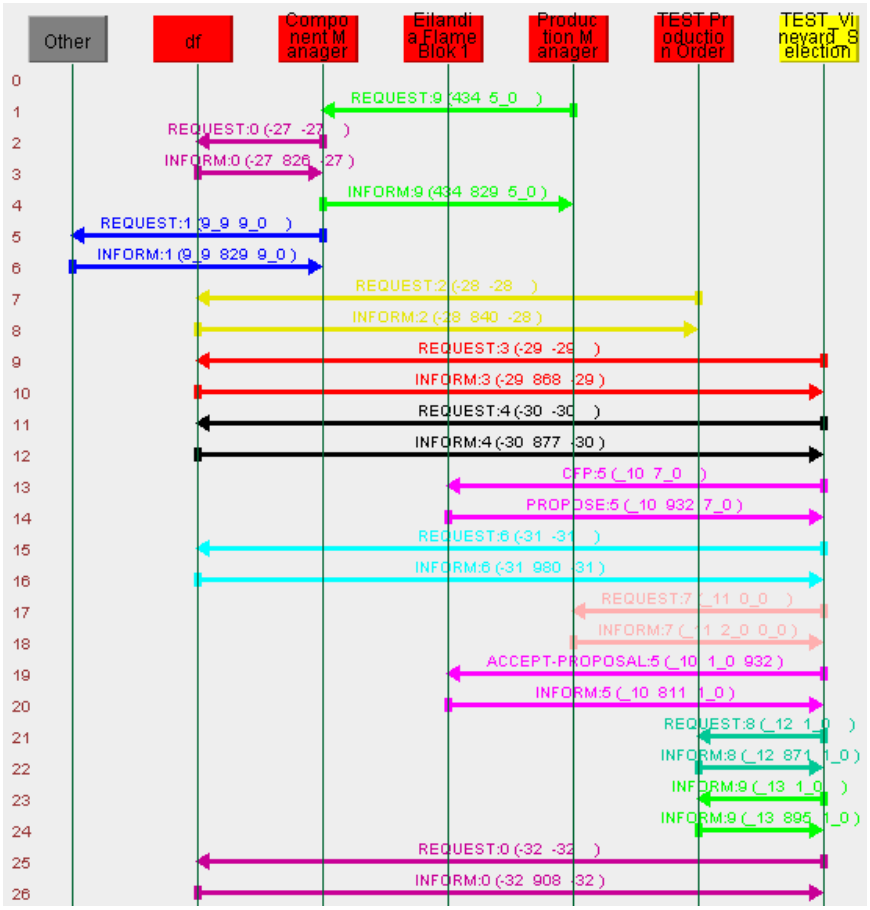


Figure 28: Sequence diagram for initiating a production order and assigning a resource.

The resource selection activity then sends the list of resources with their proposals to a decision maker (light pink lines) that is obtained from the DF (light blue lines). Once the decision maker sends the selected resources, the resource selection activity informs the resources if their proposals were accepted or rejected (pink lines). Once the resources acknowledged, the Vineyard Selection activity sends a request to the TEST Production Order as soon as a selected resource acknowledged their acceptance to the production order and the TEST Production Order acknowledges the receipt of the selected resource (green lines).

When the Vineyard Selection activity received responses from all the resources, it informs the TEST Production Order that the vineyard selection is complete and the TEST Production Order acknowledges (light green lines). The Vineyard Selection activity then removes its service from the DF (purple lines) before terminating.

6.3.2 Monitor Production Order

The DS was used to monitor a production order containing the resources specified in Table 14. The figures in this section are snippets of the full results provided in Appendix A.3.

With reference to Figure 29, the black lines represent the *request* message sent by the Production Manager to request the list available production order activities from the DF. The DF return the TEST Production order activity and the Production Manager requests its latest progress (pink lines). The TEST Production Order responds with a percentage complete value, since it keeps track of the completion of all its tasks and is complete when all the grapes are packed.

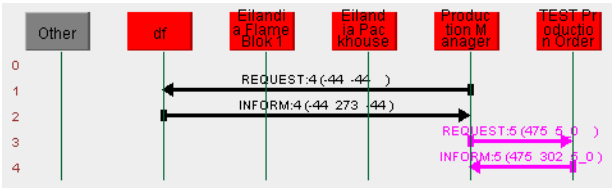


Figure 29: Sequence diagram for requesting production order progress.

With reference to Figure 30, the Production Manager can request the list of assigned resources from a production order. The purple lines represent the request sent from the Production Manager to the TEST Production Order to request its assigned resources. The TEST Production Order then responds with a list of its assigned resources. Appendix B.2 shows an example of a response from the TEST Production Order sending its list of assigned resources. The Production Manager can then request the information of a resource. The Production Manager requests the information from a vineyard resource and the resource responds by sending an inform containing its latest information (blue lines).

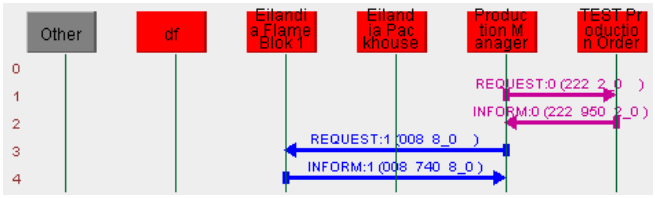


Figure 30: Sequence diagram for requesting assigned resources and assigned resource information.

With reference to Figure 31, the "Name Eilandia Harvesting Team" updates the TEST Production Order with its latest progress (pink lines) by sending a request with its latest information. The TEST production Order then updates the progress of its task before responding with the resource's latest target value. The resource updates its progress again (light blue lines), but this time the target value of the task is reached. The resource assigned to the task is then removed from the production order (light pink lines) by sending a cancel message to the resource to remove the production order from its schedule. The resource then acknowledges its removal of the production order with an *inform* message.

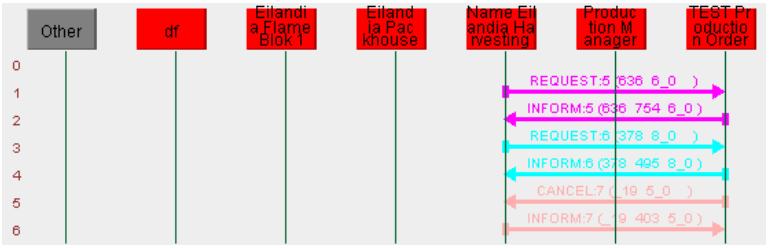


Figure 31: Sequence diagram for updating progress and reaching target value.

The packhouse manager can update the packaged grapes of a packhouse to update the completion progress of a production order. In Figure 32, the green lines represent the "Eilandia Packhouse Manager" requesting the packhouse resource from the DF. The packhouse manager resource then sends a *request* message to update the packhouse resource with the latest information before the packhouse resource acknowledges (light green lines). The "Eilandia Packhouse" then sends a request to update the TEST Production Order with its latest progress to which the production order activity responds with the new target value (purple lines).

The blue lines represent another update of information between the packhouse manager agent and the packhouse resource. The yellow lines represent the packhouse resource updating the production order activity. This time, the target value for the production order is reached which completes the production order activity. Before the production order terminates, it must remove the resources still assigned to the production order (red lines).

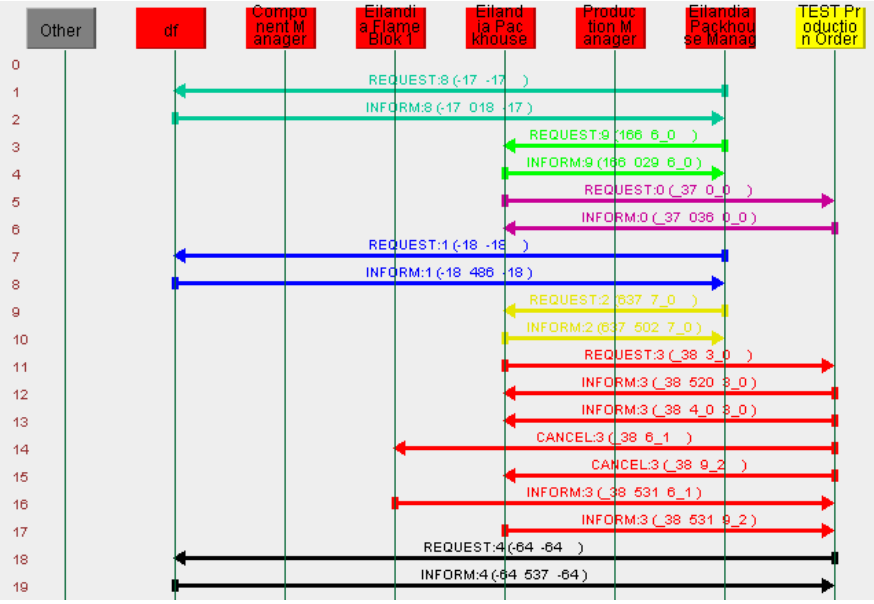


Figure 32: Sequence diagram of the packhouse manager updating the packaged pallets, reaching the target and completing the production order.

The production order informs the packhouse resource that its target value is reached before sending a *cancel* message to all its assigned resources to remove the production order from their schedule (red lines). Once the resources respond to acknowledge, the production order removes its service from the DF (black lines) before terminating. Ideally, all the assigned resources would have completed their production order tasks before the packhouse completes the production order by reaching the target value for the packaged grapes. All the assigned resources would then have been removed and the production order only needs to inform the packhouse resource to update its schedule before terminating.

With reference to Figure 33, the Production Manager requests the list of previous production orders form the Component Manager (red lines). The Production Manager sends a request to the Component Manager to retrieve the list of previous production orders from text files and responds with an inform containing the list of previous production orders. The Production Manager can then request to view the information stored on a production order (black lines). The Production Manager sends a request to the Component Manager containing the name of the production order to view its information. The Component Manager then reads the stored information from the text file and responds with an *inform* message containing the information on the requested production order. Appendix B.3 shows the list of assigned resources stored on the TEST Production Order.

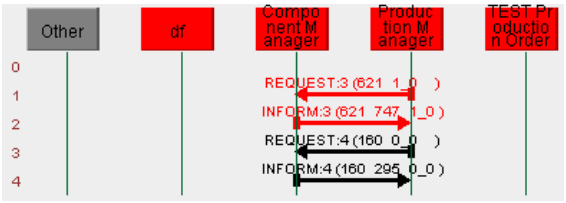


Figure 33: Sequence diagram for requesting previous production orders and a production order's information.

The farm manager resources can view the information on all the resources on the farm they manage and update the information of a vineyard or packhouse resource. In Figure 34, the light pink, green and light green lines represent the requests send to the DF to obtain the list of all the resources of a specific resource type. The farm manager then requests the farm each resource belongs to. The resources then respond with an inform containing the farm they belong to (purple lines). The resources responding with "Eilandia" all belong to the "Eilandia Farm Manager".

The farm manager can then view the information of a resource by sending a request to the resource. The resource responds with an inform containing its information (blue lines). The farm manager can then update the information of a resource by sending a request containing the updated information (yellow lines). Once the information is updated and the farm manager receives an inform, the farm manager will request the updated information from the resource (red lines)

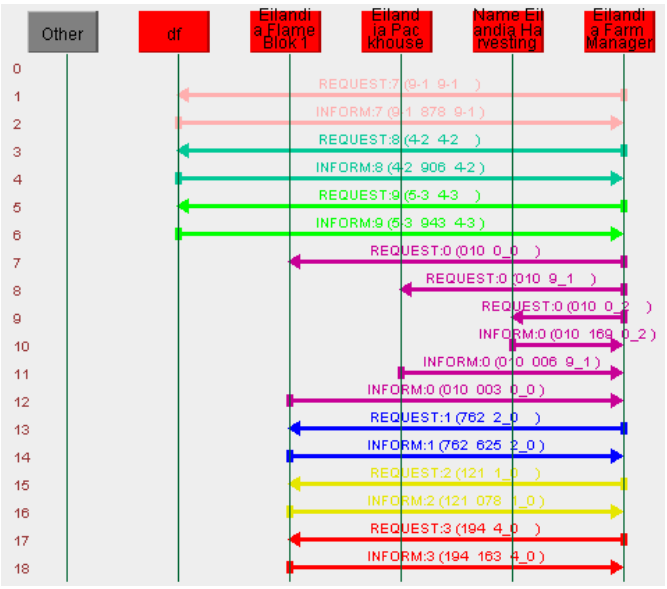


Figure 34: Sequence diagram of the farm manager requesting farm resources, requesting a resource's information and updating the resource's information.

6.3.3 Production Order Changes

The DS was used to make changes to an existing production order according to the set of changes specified in Table 14. The figures used in this section are snippets of the production order changes. The full results are provided in Appendix A.4.

A production order activity can be changed by removing an assigned resource (Figure 35). The Production Manager sends a *request* containing the resource which the production order activity must remove (blue lines). The TEST production order sends a *cancel* message to the resource that needs to be removed (yellow lines). The resource then removes the production order from its schedule before acknowledging. The Production Manager then also requests a new list of the resources assigned to the production order (red lines) by sending a request to which the TEST Production Order responds with its new list of assigned resources.

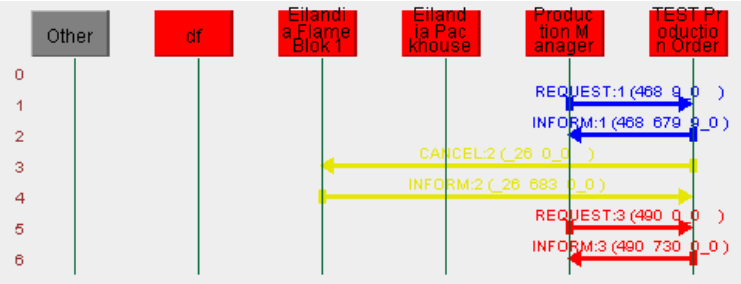


Figure 35: Sequence diagram of the production manager removing a resource from a production order and obtaining a new list of assigned resources.

The resources can remove themselves from a production order (Figure 36). The "Name Eilandia Harvesting Team" resource must terminate and can therefore no longer participate in the execution of a production order task. The harvesting team resource sends a request to its assigned production orders to request its removal from the list of assigned resources (red lines). Once the production orders have acknowledged the resource's removal, the resource sends a request to the DF to remove its service (black lines) before terminating.

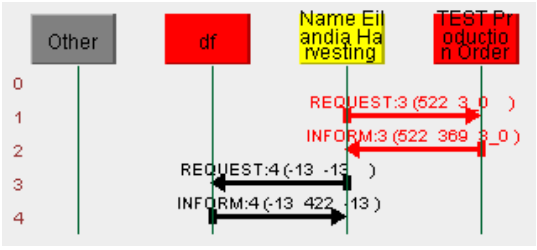


Figure 36: Sequence diagram of a resource removing itself from a production order.

A production manager can add a resource to a production order (Figure 37). The production manager sends a request to the TEST Production Order to add a resource (pink lines). The TEST Production Order then initiates a resource selection activity before acknowledging the resource addition with an inform. The resource selection activity follows the same sequence as described in section 6.3.1.

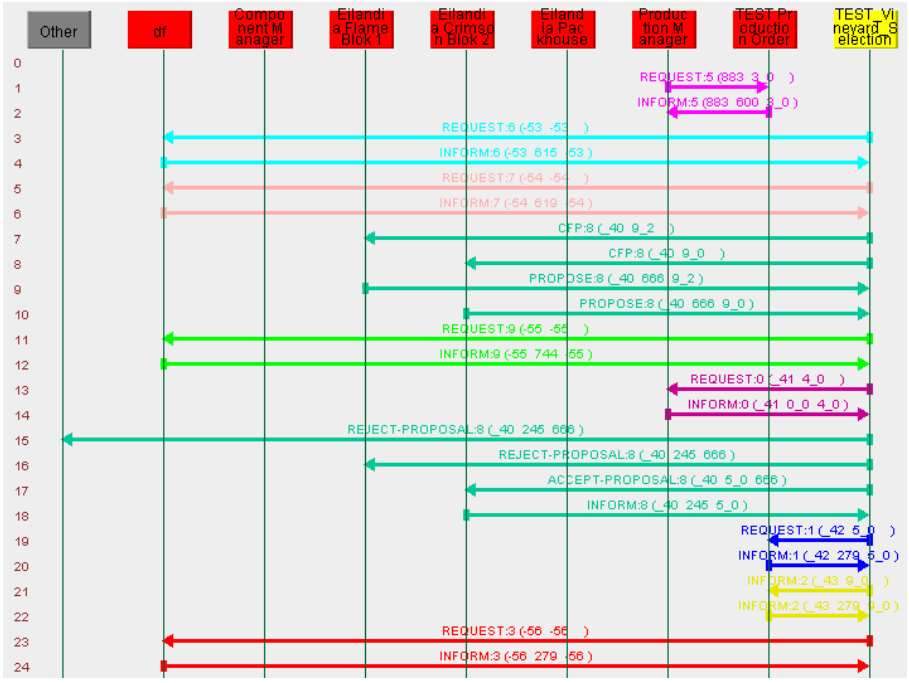


Figure 37: Sequence diagram of the production manager adding a resource to a production order.

6.3.4 Cancel Production Order

The DS was used to cancel an existing production order containing at least one of each resource type, as presented in Table 14. The result of the production order cancellation is provided in Appendix A.5. Figure 38 shows a snippet of the full result to explain the functionality of the system.

The Production Manager can cancel a production order (Figure 38). The black lines represent the messages sent by the Production Manager to the TEST Production Order to cancel the production order activity, as well as the messages sent to the assigned resources. The TEST Production Order acknowledges the request from the Production Manager with an inform message before sending a *cancel* message to the assigned resources (“Eilandia Flame Blok 1” and “Eilandia Packhouse”) to request that they remove the production order from their schedule. Once each resource acknowledged the production order’s removal, the production order removes its service from the DF (pink lines) before terminating.

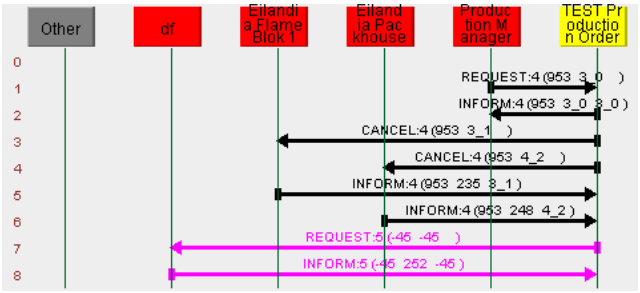


Figure 38: Sequence diagram for cancelling a production order.

6.3.5 Results Discussion

The described experiments were conducted and, from these experiments, the functionality of both systems - in terms of communication, information management and decision support - were examined and are discussed in the following sections.

6.3.5.1 Communication

The experiment results of the communication accuracy of both systems are provided in Table 15. Similarly, the communication reliability results are provided in Table 16. A discussion on the results follow the tables.

Table 15: Communication accuracy results

	Message Exchanges	Alteration Possibility	Message Format
CS	2	Every exchange	Unstructured
DS	1	No	Structured

The relevant information during the initiation of a production order needs to be communicated to the relevant resources. In the CS, the production manager sends a message to inform the farm manager, who then forwards the necessary information to the farm resources. Therefore, the message must be exchanged from production manager to farm manager, and from farm manager to resource. In contrast, the DS sends the message directly from the production manager to the resources, reducing the number of message exchanges, which reduces the chance of a message being altered or not reaching the intended receiver.

When the production manager uses the DS to send a message, the message is sent directly to the receiver and cannot be altered. When a resource is assigned with the CS, the production manager sends a custom message to the farm manager. The farm manager then sends a message or verbally informs the resources of their instructions. The farm manager can interpret and alter the messages before forwarding the messages to the resources. This reduces the system's reliability.

The DS uses specific XML formats to structure the information when sending messages. These formats cannot be altered and stays consistent for each type of message. Currently, the production and farm managers are free to construct the messages as they see fit and can choose the platform to send the messages. This provides more room for misinterpretations reducing the system's reliability.

Table 16: Communication reliability results

	Message Exchanges	Message Receipt Acknowledgement	Receiver Availability
CS	2	No guaranteed	Manually request
DS	1	Guaranteed	System keeps track

The DS uses a request-inform message protocol. When a message is sent, the receiver responds to acknowledge the receipt of the message. This ensures that the system, and sender, knows the message is received. With the CS, the receiver will receive the message and not acknowledge the receipt of the message. Due to the nature of the system, the sender will often not know whether the message is received by the intend receiver. The receiver can also decide what to do with the received message. The receiver can either read the message or not, or the receiver can decide to respond to the message or not. Therefore, the DS is more reliable.

The DS requires that resources register their service when they are active and available. This allows decision makers to obtain the available resources when selecting resources to assign to a production order to prevent assigning resources that are currently unavailable to perform a service. With the CS, the production manager sits in an office for most of the time and must manually send messages to personnel to enquire about active resources, before initiating production orders or assigning resources. Therefore, the DS is more reliable.

6.3.5.2 Information Management

The experiment results of the information management accessibility of both systems are provided in Table 17. Similarly, the information management traceability results are provided in Table 18. A discussion on the results follow the tables.

Table 17: Information management accessibility results

	Information request	Number of messages	Closeness to real time
CS	Manual	Depends on number of resources	Depends on frequency of requests
DS	Automatic	1	Specified update period

The production manager needs to know the progress of the production orders and how each assigned resource is performing to proactively adjust the production orders. The DS updates the information provided to the production manager according to a specified update period. The updates consist of the active production orders and their completion progress. The production manager then has the option of examining each production order and its assigned resources. The DS also allows the farm manager to examine each of its farm's resources.

With the current PM system, the production manager needs to send messages to manually request the updated information. The progress and information on each resource must be requested individually. This causes latency since each request takes time and the resources might not have the required information readily available. However, the CS has more freedom when it comes to requesting information. There is no structured format when requesting information that restricts the information that can be requested.

When the latest information on resources are required, the DS obtains this information from the resources and presents the information to whoever requested the information. With the CS, the information needs to be requested manually by sending a message to each resource to obtain its information. Therefore, the CS requires more messages to be sent to obtain the information. The time required to obtain the information is subject to the response time of the receiver. The DS acquires the information directly from the resource without a delay induced by the resource's response time. This makes the information of the DS more accessible.

The DS updates the information presented to production managers about the production orders and their completion progress according to a specified update period. With the CS, the production manager has more freedom when requesting information. However, the production manager is unable to have the information on all the active production orders as frequently updated and available as provided by the DS. This makes the instance-specific information on the production orders more accessible with the DS.

Table 18: Information management traceability results

	Information Stored	Information Structure	Information Accessibility	Person who can access
CS	Person Memory	None	Request from person	Human memory dependent
DS	Text File	XML	Multiple devices	Anyone requiring the information

The DS stores the information about production orders locally within each activity and resource entity until the entity terminates. Therefore, the production order activity stores the assigned resources and their information in a text file in XML format. The production and farm managers can use this information to improve future resource assignments to production orders.

The CS do not have a dedicated storage for the information about production orders. The CS relies on human memory to remember the resource assignments and their execution information. The messages sent can be reviewed if they were not cleared or deleted, but it is impractical to try and find the correct message among all the messages being sent daily. Therefore, the DS has better traceability with regards to storing information.

The information which the DS stored in text files can be read and sent to any device requesting the information. The CS relies on human memory to store information and can be accessed by anyone who requests the information from the person who remembers the desired information. The information can only be accessed when the person is reachable by either verbally requesting the information in person or by sending a message. Therefore, the DS stores information more securely and is accessible on multiple devices, improving the traceability.

6.3.5.3 Decision Support

The experiment results of the decision support consistency of both systems are provided in Table 19. Similarly, the decision support agility results are provided in Table 20. A discussion on the results follow the tables.

Table 19: Decision support consistency results

	User Decision-Making Permissions	Decision-Making Process	Provided Information
CS	Anyone with authority	No process	Varying personal knowledge
DS	Decision makers	Structured	Repeatedly gathered same resource information

The DS restricts the decision making to designated decision makers, allowing the same type of decisions to be made consistently by the same people. The production manager makes production order decisions and the farm managers make farm decisions. The CS does not restrict decision making. Anyone with authority can make decisions and make changes to assigned resources, e.g. a production manager can make farm manager decisions. Restricting decision making to designated decision makers improve the decision-making consistency.

The DS restricts decision making to dedicated decision makers, allowing the system to ensure that all the resources and relevant personnel will be informed of the result of a decision. The CS does not enforce this restriction. This may lead to resources and relevant personnel not being informed of decisions. The CS can achieve quick responses by allowing any person with authority to act when a change needs to be made. However, this does not improve the agility of the system, since the changes do not necessarily reach all the relevant resources.

The DS enforces a structured decision-making process, where information is gathered on the resources before a decision can be made. The CS does not follow such a structured process. A decision is often made in the moment with the instantaneous knowledge about the production order. This also leads to decisions not being communicated to everyone affected by the decision. This leads to the best decisions not always being made and not reaching everyone affected. Therefore, a structured process can improve the consistency of decisions by providing information to make informed decisions. It also improves the agility by ensuring the result of decisions is communicated to everyone affected.

The DS obtains the available resources and their information relevant to determine their suitability and availability to a production order. With the CS, the decision maker can request the relevant information from a resource to make an informed decision. However, the response from the resource is often subjective and the response time is dependent on the resource. This results in uninformed decisions being made and reduces the consistency by making decision without the same type of information available for each decision.

Table 20: Decision support agility results

	Provide Decision-Making Information	**Message Conveying**
CS	Request manually	Manually send messages
DS	Gathers resource information periodically	System sends messages

The DS periodically updates the information on production orders and their progress. This allows to the decision makers to have up to date information and examine each production order to know when to make changes. With the CS, the decision makers need to acquire information manually. Additionally, they depend on their own intuition when to acquire information to determine if production order changes are required. Providing information to support decision makers to decide when to make changes improves the agility of the system.

The DS follows a structured process where the result of decisions is communicated to all the relevant resources. The resources acknowledge the receipt thereof. The CS relies on the decision makers to inform everyone affected. This could lead to some resources not being informed. Therefore, the DS has a higher agility by structuring the way the resources are informed of the result of decisions.

6.3.5.4 Discussion

The DS was able to perform the required functionalities, presented in Table 14, for each experiment. From the results of the experiments, the DS can improve on the current table grape PM system with regards to communication, information management and decision support. Based on the results presented in section 6.3.5.1, the communication can be improved in terms of the accuracy and reliability. Based on the results presented in section 6.3.5.2, the information management can be improved in terms of the accessibility and traceability. Based on the results presented in section 6.3.5.3, the decision support can be improved in terms of the consistency and agility.

Furthermore, the DS system was able handle frequent and sudden changes regarding changes to production orders, resulting in a flexible and adaptable system. The DS was also robust against disturbances, and the decision makers could alter the production orders and assigned resources whenever a disturbance occurred, e.g. a resource removing itself from a production order. The improvements made by the DS regarding providing decision makers with information, along with informing resources of the decisions, enables the system to make changes efficiently and effectively.

6.4 Benefits and Drawbacks of the ARTI Holonic Architecture Implementation

The ARTI reference architecture specifies that execution functionalities be encapsulated in the IBs and decision-making functionalities be encapsulated in the IAs. This eases the mapping of functionality and the reconfiguration, by separately and independently making changes to a decision making or execution component. It also allows the developer to focus on implementing the individual components to ensure that each component is implemented correctly. However, the distribution of functionality across the IB and IA components makes the implementation more challenging.

The Instance IA components that are generic and can be used for all types of activities or resources, together with the Type IA encapsulating activity or resource specific decision making, enables the development of a generic Instance IB that can perform all execution. Additionally, this enables the Instance IB to execute the NEU protocol, since the Type IA determines what behaviours to execute when. The implementation of the NEU protocol is beneficial in terms of reconfigurability. The NEU protocol enables that either execution or decision-making components be changed to change the execution of the agent, without having to make changes to how the agent is initiated and how it interacts with its relevant ARTI components.

The IB components reflect reality and provide real-time information. The IA is a separate component that is responsible for the decision making. Implementing the decision making as agents allows decision-making algorithms to execute concurrently with the execution in the IB. This allows for the creation of DTs using the real-time, reality-reflecting IB components.

The generic terminology of ARTI eases the mapping of system components by distinguishing between service providing and service coordinating elements, as well as distinguishing between execution and decision-making functionality. This simplifies the implementation of a holonic system.

Multiple resource holons monitoring and managing their own small environment reduced the complexity of the system and enables dynamic addition and removal of resources or activities. Each resource holon monitoring and managing its own small environment, together with the inherit cooperation of holons, enables the system to have real-time information on system processes and all the tasks each system process consists of.

ARTI makes provision for humans performing tasks. JADE-LEAP enabled the development of an Android application to integrate human workers with the system. JADE and JADE-LEAP eases the implementation of a distributed system using an Android application by providing aids such as the Java libraries, O2A interface and broadcasting intents. However, a distributed system using JADE and JADE-LEAP still requires network coverage to create a distributed ARTI-based system. Since some agents run on a PC and others on mobile devices, the component manager agent was required to allow a mobile device to create an agent on a PC.

JADE provides debugging tools, such as the sniffer agent, that eases the debugging and development. JADE also provides clear explanations of the behaviours that are simplified to ease the development of a simple agent-based system. However, development in JADE becomes challenging when trying to implement more complex behaviours. A thorough understanding of the behaviours is required to manipulate them to achieve complex functionalities, e.g. the NEU protocol that needs to execute behaviours obtained from the IB type component.

7 Conclusion and Recommendations

The ARTI reference architecture requires that the system elements be mapped to activities or resources. This makes it possible to reduce the complexity of a systems by breaking the system down into autonomous and cooperative holon entities. The ARTI reference architecture further reduces the complexity by separating the execution and decision-making functionalities. The developer can separately focus on optimizing the execution or decision-making functionalities without having to make changes to the other functionalities. The ARTI reference architecture also divides each execution or decision making component into types and instances. This allows the developer to change the execution or decision making functionality encapsulated in the types, without changing the interaction functionality between components encapsulated in the instances.

The table grape PM was examined to identify the elements that need to be mapped and implemented according to the ARTI reference architecture. Each of the identified elements were classified as either a resource or activity. The resources perform some service and were identified as: vineyards, packhouses, pack material store, harvesting teams, quality control stations, packing stations, transportation vehicles, production manager, farm managers and packhouse managers. The activities coordinate the performance of services and were identified as: production orders and resource selection activities. The activities and resources were further divided into IBs and IAs, as well as instances and types. The Instance IB components of both resources and activities only contain the functionality to perform the NEU protocol. The Type IB components contain the resource or activity specific execution functionality. The Instance IA components of both the resources and activities contain the decision-making functionality to direct the Instance IB to the Type IA containing the resource or activity specific decision-making functionality.

The implementation of the table grape PM system was done using JADE and JADE-LEAP. JADE provides a development framework to create a MAS with agents using behaviours to perform their tasks. JADE also provides graphical tools to aid in the development and debugging of such a system. JADE-LEAP enabled the development of distributed system using an Android application on mobile devices to interface with the resources consisting of human workers.

The functionality and performance of the DS was evaluated by comparing the DS to the CS. The criteria that was used consist of the communication, information management and decision support. The communication is evaluated in terms of accuracy and reliability. The information management is evaluated by considering the accessibility and traceability of the system's information. The decision support offered by the system is evaluated in terms of consistency and agility.

From the discussion on the results in section 6.3.5.4, the DS can improve the table grape PM with regards to communication, information management and decision support. The communication accuracy is improved by increasing the certainty that the end receiver will receive the message as intended by the sender. The communication reliability is improved by increasing the certainty that the system will consistently deliver the messages to the intended receiver and that action will be taken accordingly. The information management accessibility is improved by obtaining dynamic, instance-specific information on active production orders and communicating this information to the relevant personnel. The information management traceability is improved by storing and recalling stored information on current and previous production orders. The decision support consistency is improved by ensuring decisions are made consistently by the same type of decision makers, with the same information available. The decision support agility is improved by supporting changes to previous decisions, e.g. decisions related to resource assignments.

It is recommended that future research focus on expanding the decision-making functionality in the IA components. These components can be implemented as agents, allowing them to be autonomous and cooperative. This will enable the IA components to run decision-making algorithms to optimize the execution of system tasks concurrently with the execution IB components. The optimization of system tasks can be achieved by allowing the IA components to make decisions regarding the schedule of the IB components to optimize the execution of production orders.

This thesis evaluated the DS on the resources and activities identified in section 4.1. Future research can expand the scope by including resources and activities on more farms, as well as include aspects beyond the production, e.g. the irrigation of the vineyard. This can also be used to evaluate the scalability of the system in terms of the number of resources and activities, as well as the capacity of the system components to send and receive messages.

The performance of the DS can be further evaluated by using the system in a real-world table grape PM environment. This will allow the system to be evaluated in terms of practicality and user friendliness, as well as further exploring the performance and scalability of the system.

The DS is dependent on workers to update the progress information of resources. This induces human error on the system. To make the system more reliable, future research can focus on implementing sensors to gather information and update the progress of resources, e.g. adding barcodes to the carton boxes that can be scanned to keep track of the packaged grapes.

8 References

Ali, O., Valckenaers, P., Van Belle, J., Germain, Saint, B., Verstraete, P. & Van Oudheusden, D., 2012. Towards online planning for open-air engineering processes. *Computers in Industry,* Volume 64, pp. 242-251.

Amarnath, G., Simons, G.W.H., Alahacoon, N., Smakhtin, V., Sharma, B., Gismalla, Y., Mohammed, Y. & Andriessen, M.C.M., 2018. Using smart ICT to provide weather and water information to smallholders in Africa: The case of the Gash River Basin, Sudan. *Climate Risk Management,* Volume 22, pp. 52-66.

Aqeel-ur-Rehman, Abbasi, A. Z., Islam, N. & Shaikh, Z. A., 2014. A review of wireless sensors and networks' applications in agriculture. *Computer Standards & Interfaces,* Volume 36, pp. 263-270.

Bellifemine, F., Caire, G. & Greenwood, D., 2007. *Developing multi-agent systems with JADE.* West Sussex, England: John Wiley & Sons Ltd.

Bergenti, F., Caire, G. & Gotta, D., 2014. Agents on the Move: JADE for Android Devices. *CEUR Workshop Proceedings,* Volume 1260.

Bhusal, S., Bhattarai, U. & Karkee, M., 2019. Improving pest bird detection in a vineyard environment using super-resolution and deep learning. *6th IFAC Conference on Sensing, Control and Automation for Agriculture,* Volume 52, pp. 18-23.

Borangiu, T., Oltean, E., Răileanu, S., Anton, F., Anton, S. & Iacob, I., 2020. Embedded digital twin for ARTI-type control of semi-continuous production processes. *Proceedings of SOHOMA 2019,* Volume 853, p. 113–133.

Borangiu, T., Oltean, V.E., Raileanu, S., Voinescu, I.L., Anton, S. & Anton, F., 2020. Modelling service processes as discrete event systems with ARTI-Type holonic control architecture. *Proceedings of 10th International Conference, IESS 2020,* Volume 377, pp. 377-390.

Bussmann, S., 1998. An agent-oriented architecture for holonic manufacturing control. *Proccedings of the 1st International Workshop on Intelligent ManufacturingSystems,* pp. 1-12.

Caire, G. & Pieri, F., 2011. *LEAP User Guide.* s.l.:Free Software Foundation.

Cardin, O., Castagna, P., Couedel, D., Plot, C., Launay, J., Allanic, N., Madec, Y. & Jegouzo, S., 2020. Energy-aware resources in digital twin: the case of injection moulding machines. *Proceedings of SOHOMA 2019,* Volume 853, pp. 183-194.

Chirn, J. & McFarlane, D.C., 2000. A Holonic Component-Based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the 11th International Workshop on Database and Expert Systems Applications,* Volume 10, pp. 219-223.

Foit, K., Banaś, W., Gwiazda, A. & Hryniewicz, P., 2017. The comparison of the use of holonic and agent-based methods in modelling of manufacturing systems. *IOP Conference Series: Materials Science and Engineering,* Volume 227.

Kagermann, H., Wahlster, W. & Helbig, J., 2013. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0.* s.l.:National Academy of Science and Engineering.

Koestler, A., 1967. *The ghost in the machine.* 1st ed. London: Hutchinson.

Kotze, M. J., 2016. *Master's thesis: Modular control of a reconfigurable conveyor system.* Stellenbosch: Stellenbosch University.

Kritzinger, D., 2020. *Modulêre kursus in tafel- en droogdruifverbouing (Modular course on table and dried grape cultivation),* Somerset West, South Africa: Agrimotion.

Kruger, K. & Basson, A., 2013. Multi-agent systems vs IEC 61499 for holonic resource control in reconfigurable systems. *Procedia CIRP,* Volume 7, pp. 503-508.

Kruger, K. & Basson, A., 2016. Erlang-based control implementation for a holonic manufacturing cell. *International Journal of Computer Integrated Manufacturing,* Volume 30, pp. 1-12.

Kruger, K. & Basson, A., 2018. *JADE multi-agent system holonic control implementation for a manufacturing cell.* Internal technical report, Stellenbosch University

Leitao, P.J.P., 2004. *Ph.D. dissertation: An agile and adaptive holonic architecture for manufacturing.* Portugal: University of Porto.

Lin, Y. & Chen, S., 2019. Development of navigation system for tea field machine using semantic segmentation. *6th IFAC Conference on Sensing, Control and Automation for Agriculture,* Volume 52, pp. 108-113.

Mahant, M., Shukla, A., Dixit, S. & Patel, D., 2012. Uses of ICT in agriculture. *International Journal of Advanced Computer Research,* Volume 2.

Meng, F., Tan, D. & Wang, Y., 2006. Development of agent for reconfigurable assembly system with JADE. *Proceedings of the 6th World Congress on Intelligent Controland Automation,* Volume 2, pp. 7915-7919.

Meng, F., Tan, D. & Wang, Y., 2006. Development of agent for reconfigurable assembly system with JADE. *Proceedings of the 6th World Congress on Intelligent Control,* Volume 2, pp. 7915-7919.

Mohanraj, I., Ashokumar, K. & Naren, J., 2016. Field monitoring and automation using IOT in agriculture domain. *Procedia Computer Science,* Volume 93, p. 931 – 939.

Pach, C., Berger, T., Bonte, T. & Trentesaux, D., 2014. ORCA-FMS: a dynamic architecture for the optimized and reactive. *Computers in Industry,* Volume 65, pp. 706-720.

Perea, R.G., García, I.F., Arroyo, M.M., Díaz, J.A.R., Poyato, E.C. & Montesinos, P., 2017. Multiplatform application for precision irrigation scheduling in strawberries. *Agricultural Water Management,* Volume 183, pp. 194-201.

Redelinghuys, A.J.H., Kruger, K. & Basson, A., 2020. A six-layer architecture for digital twins with aggregation. *Proceedings of SOHOMA 2019,* Volume 853, pp. 171-182.

Rossello, N.B., Carpio, R. F., Gasparri, A. & Garone, E., 2019. A novel observer-based architecture for water management in large-scale (hazelnut) orchards. *6th IFAC Conference on Sensing, Control and Automation for Agriculture,* Volume 52, pp. 62-69.

Rossouw, F., 2019. *Table grape production management* [Interview] (16 June 2019).

Sihlobo, W., 2019. SA horticulture is blooming, but there's still room for growth. *Daily Maverick.*

Singgih, M.L., 2014. Holonic manufacturing system: current development and future applications. *International Conference on Engineering and Technology Development.*

Smith, R.G., 1980. The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Transactions on Computers,* Volume C-29, pp. 1104-1113.

South African Department of Agriculture, Forestry and Fisheries, 2012. *Production guideline – grapes.* [Online]
Available at: https://www.nda.agric.za/docs/Brochures/grapesprod.pdf
[Accessed 23 February 2020].

Star South, 2019. *Star South packing guide,* Wellington, South Africa: Star South.

Valckenaers, P., 2018. ARTI reference architecture – PROSA revisited. *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing,* pp. 1-19.

Valckenaers, P., 2020. Perspective on holonic manufacturing systems: PROSA becomes ARTI. *Computers in Industry,* Volume 120.

Valckenaers, P. & De Mazière, P.A., 2015. Interacting computing processes for evolvable execution systems: the NEU protocol. *Industrial Applications of Holonic and Multi-Agent Systems,* Volume 9266, pp. 120-129.

Valckenaers, P. & Van Brussel, H., 2015. *Design for the unexpected.* 1st ed. Oxford: Elsevier.

Van Brussel, H., Wyns, J., Valckenaers, P., Bongaerts, L. & Peeters, P., 1998. Reference architecture for holonic manufacturing systems: PROSA. *Computers In Industry,* Vol. 37(Special issue on intelligent manufacturing systems), pp. 255 - 276,.

Xie, J. & Liu, C., 2017. Multi-agent systems and their applications. *Journal of International Council on Electrical Engineering,* Volume 7, pp. 188-197.

Xie, W., Wang, F. & Yang, D., 2019. Research on carrot grading based on machine vision feature parameters. *6th IFAC Conference on Sensing, Control and Automation for Agriculture,* Volume 52, pp. 30-35.

Xu, L. & Weigand, H., 2001. The evolution of the contract net protocol. *Lecture Notes in Computer Science,* Volume 2118, pp. 257-266.

Appendix A Tests Sniffer Screenshots

A.1 Initiate Production Order

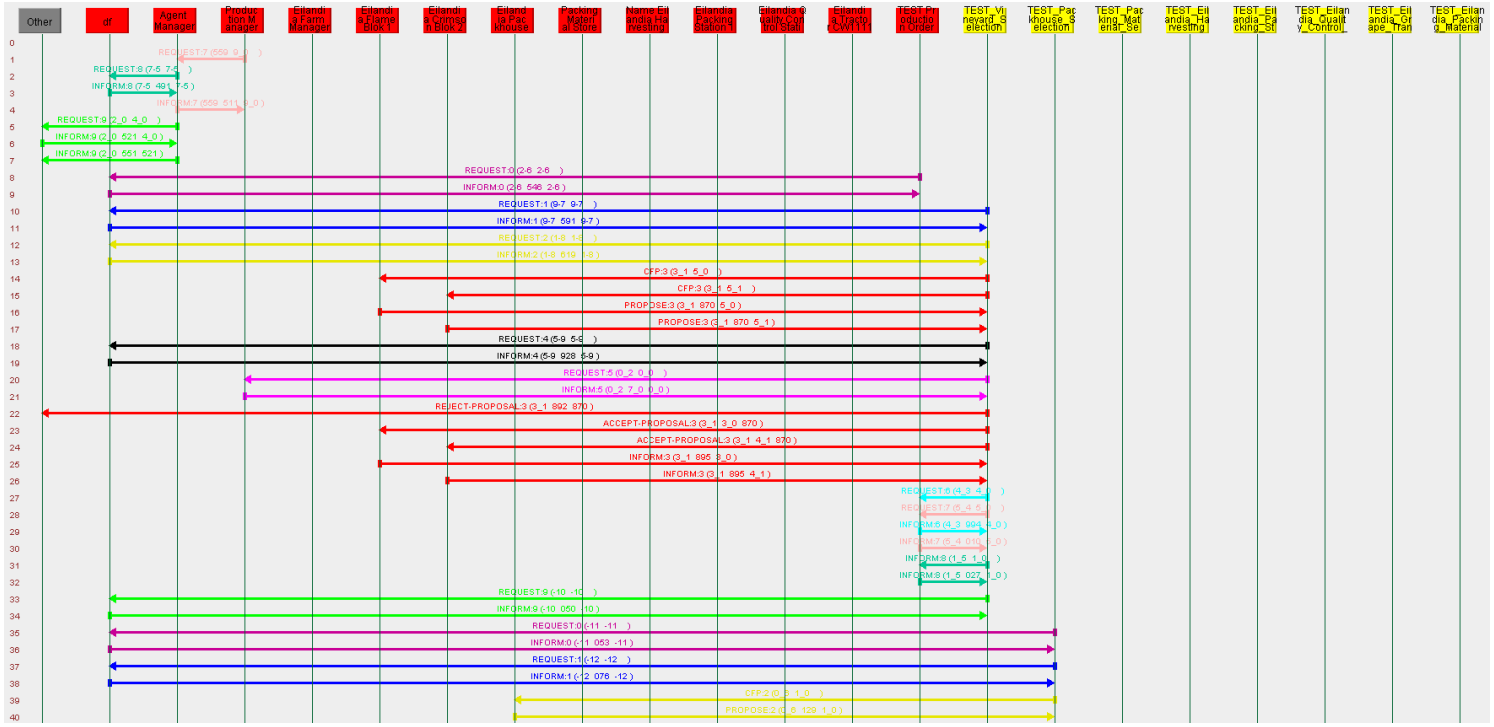


Figure 39: Initiate production order first set of messages.

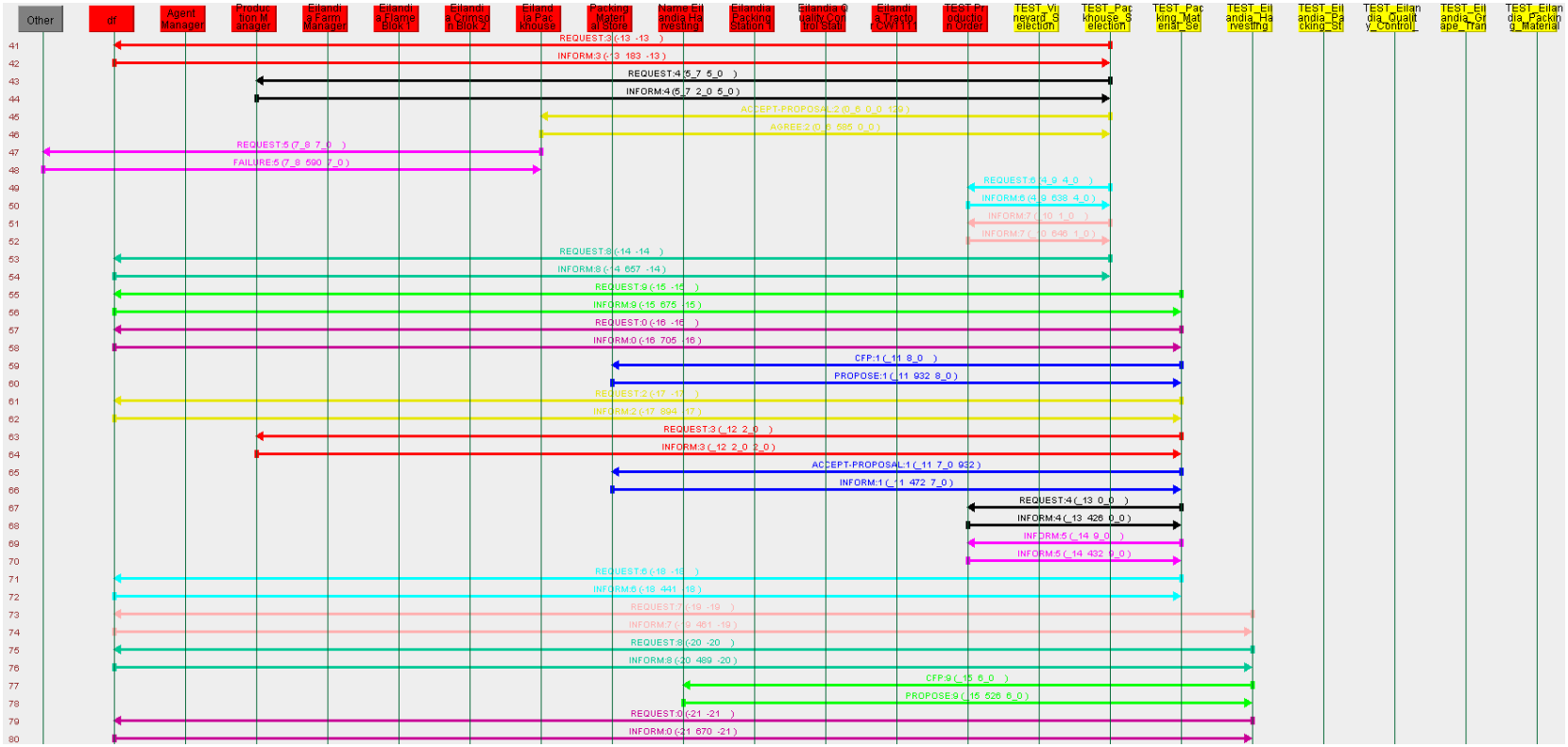


Figure 40: Initiate production order second set of messages.

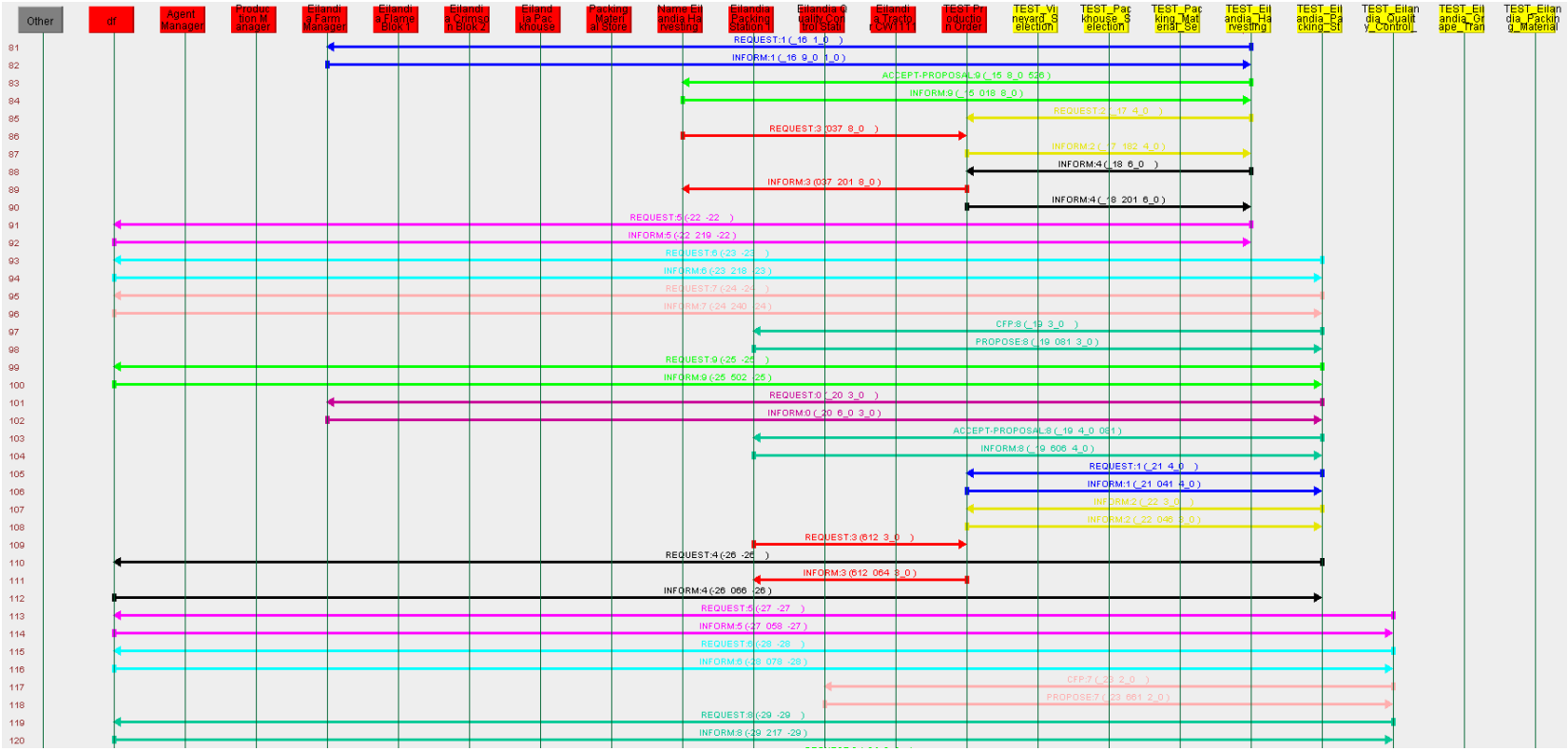


Figure 41: Initiate production order third set of messages.

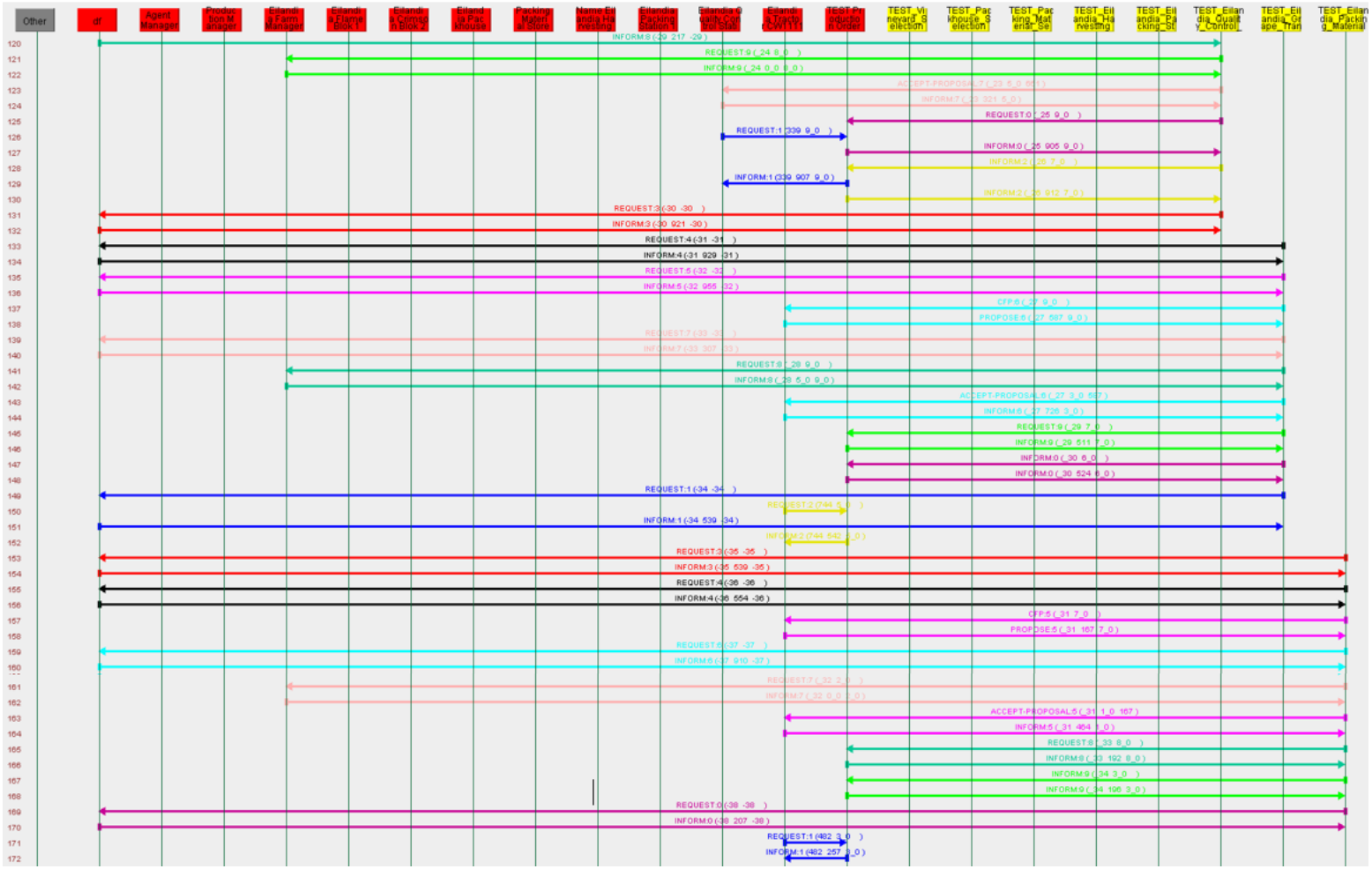


Figure 42: Initiate production order final set of messages.

A.2 Initiate Production Order with Limited Resources

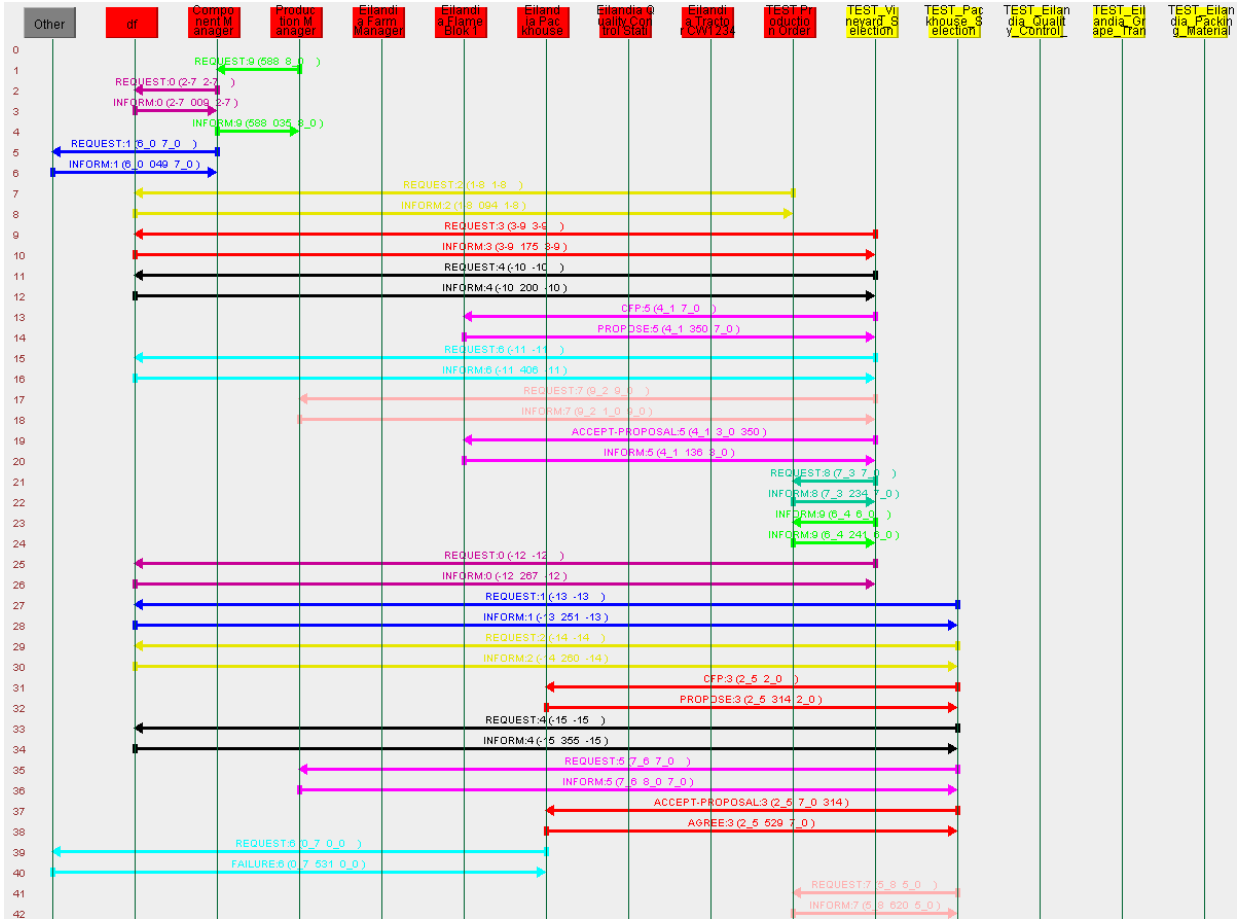


Figure 43: Initiate production order with limited resources first set of messages.

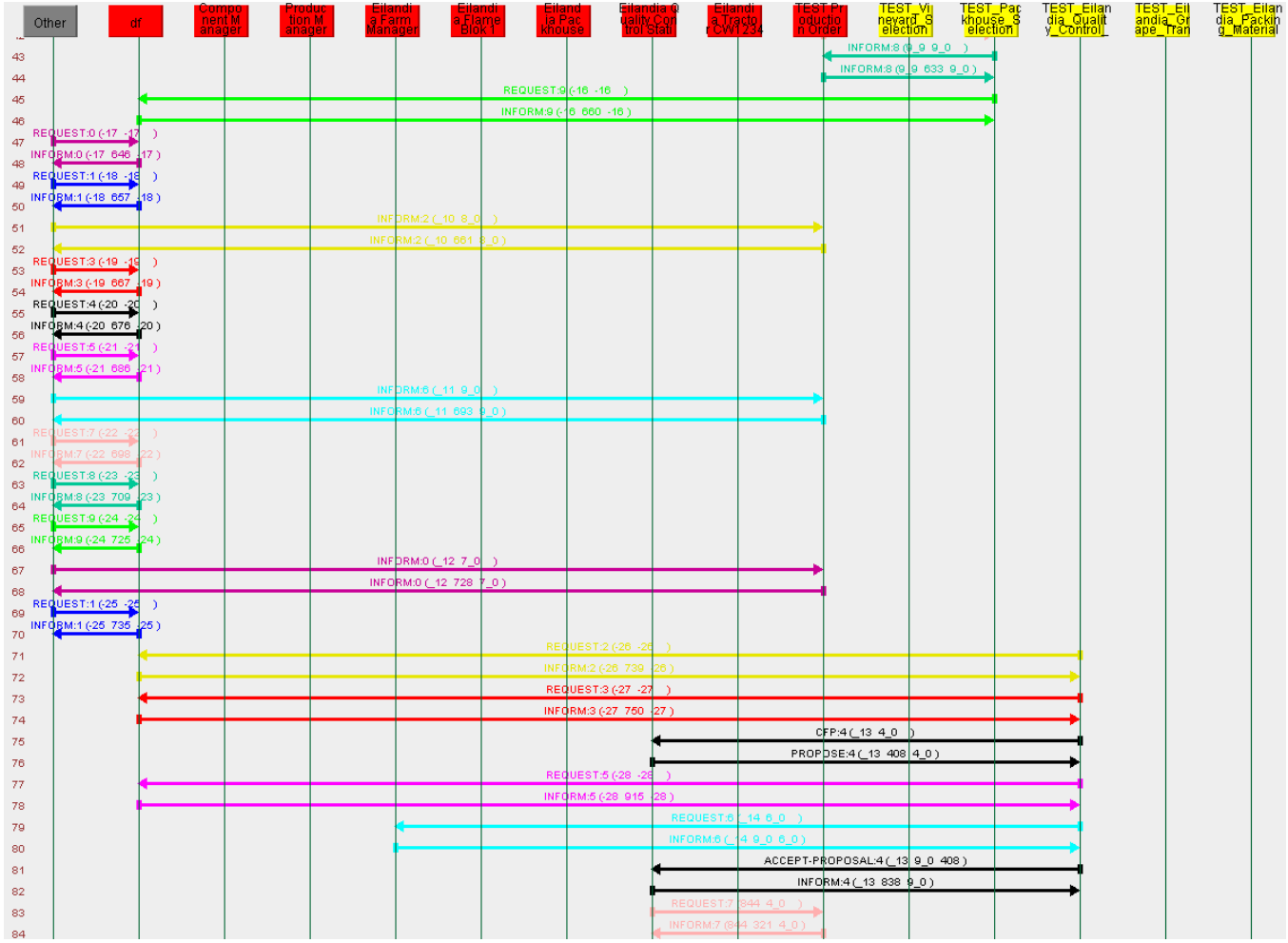


Figure 44: Initiate production order with limited resources second set of messages.

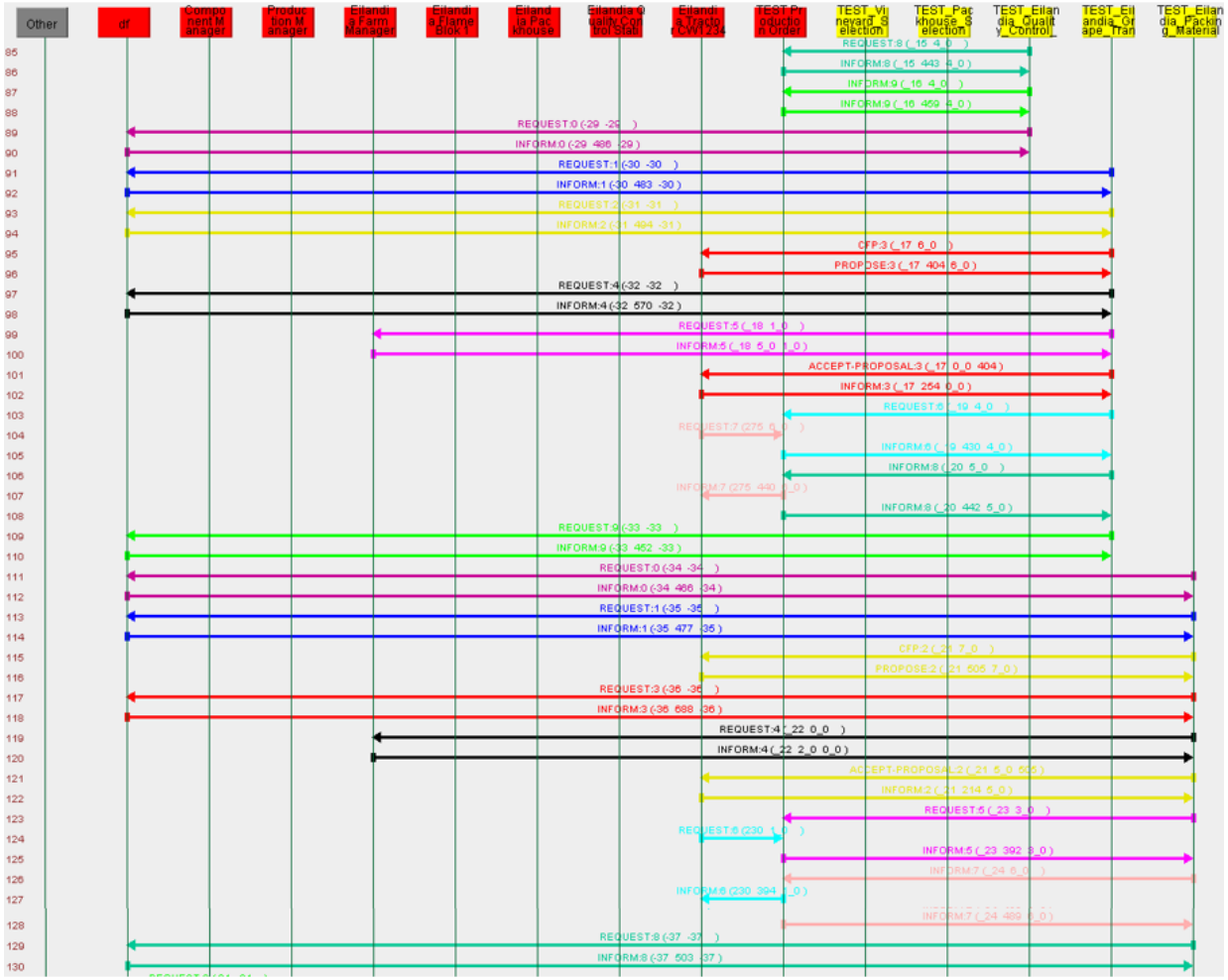


Figure 45: Initiate production order with limited resources final set of messages.

A.3 Monitor Production Order

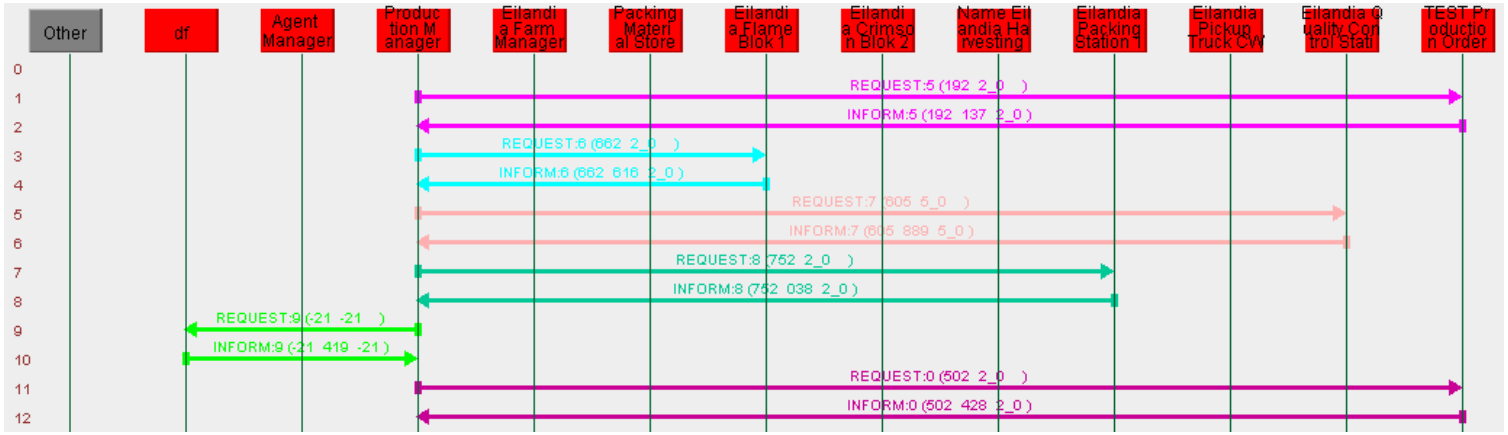


Figure 46: Production manager requesting a production order's assigned resources, an assigned resource's information and a production order's progress update.

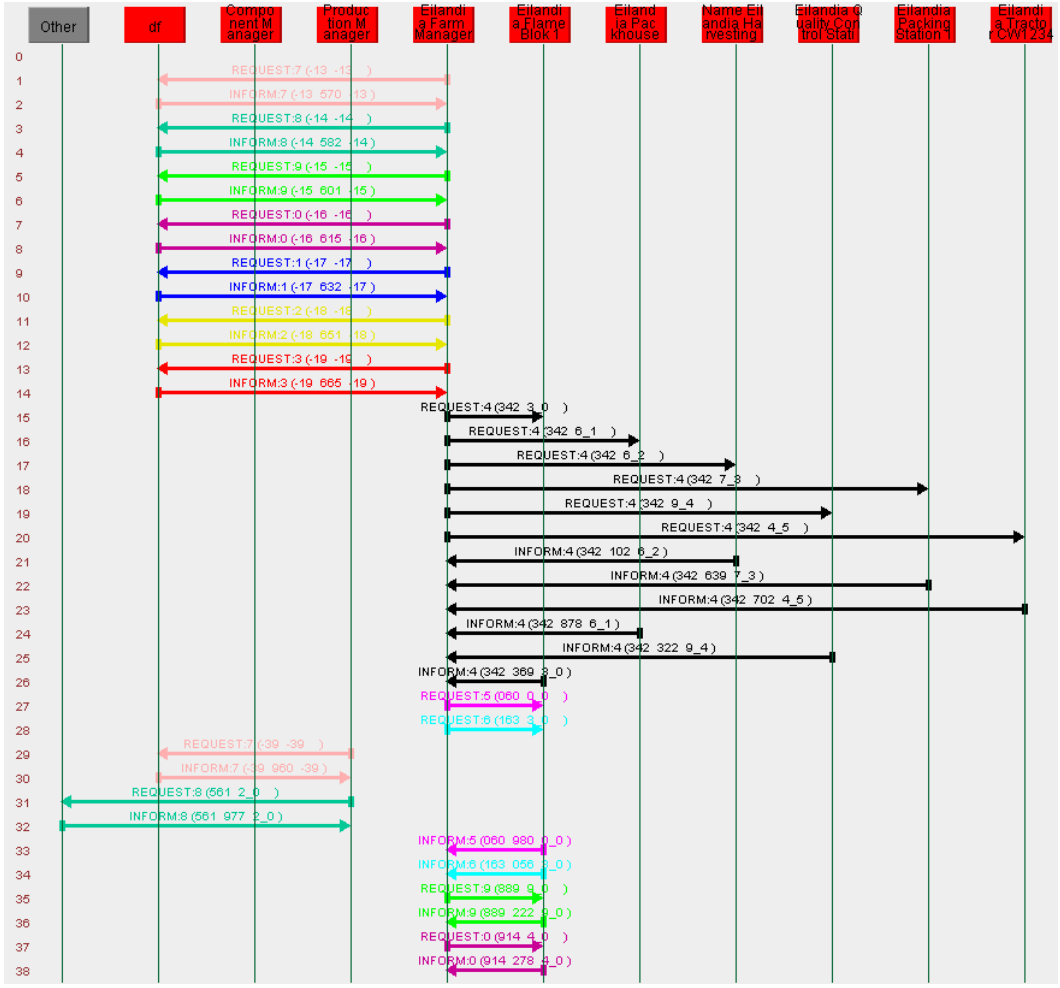


Figure 47: Farm manager retrieving its farm's resources, a resource's information and updating a resource's information.

A.4 Production Order Changes

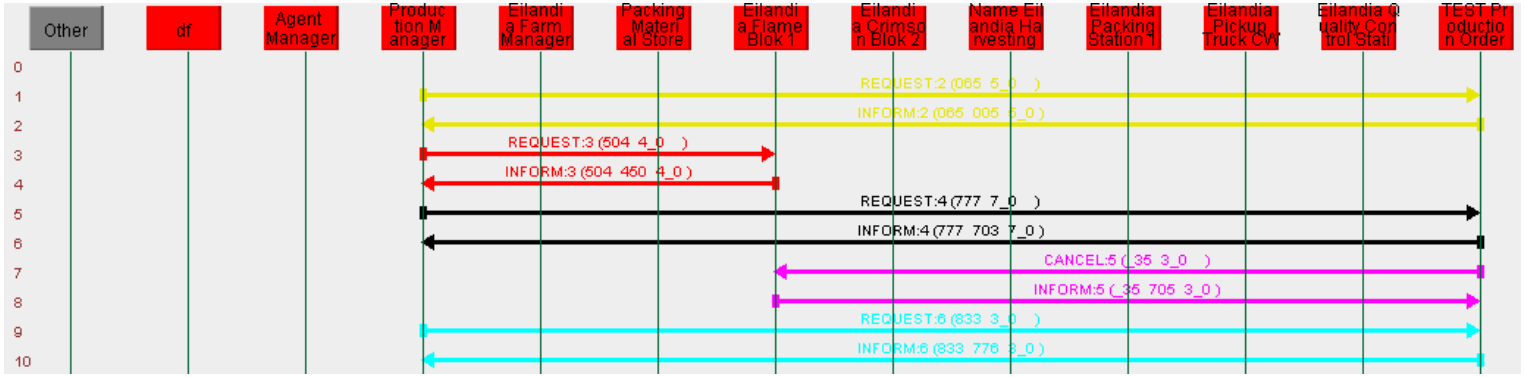


Figure 48: Request available resources and remove a vineyard resource from the production order.

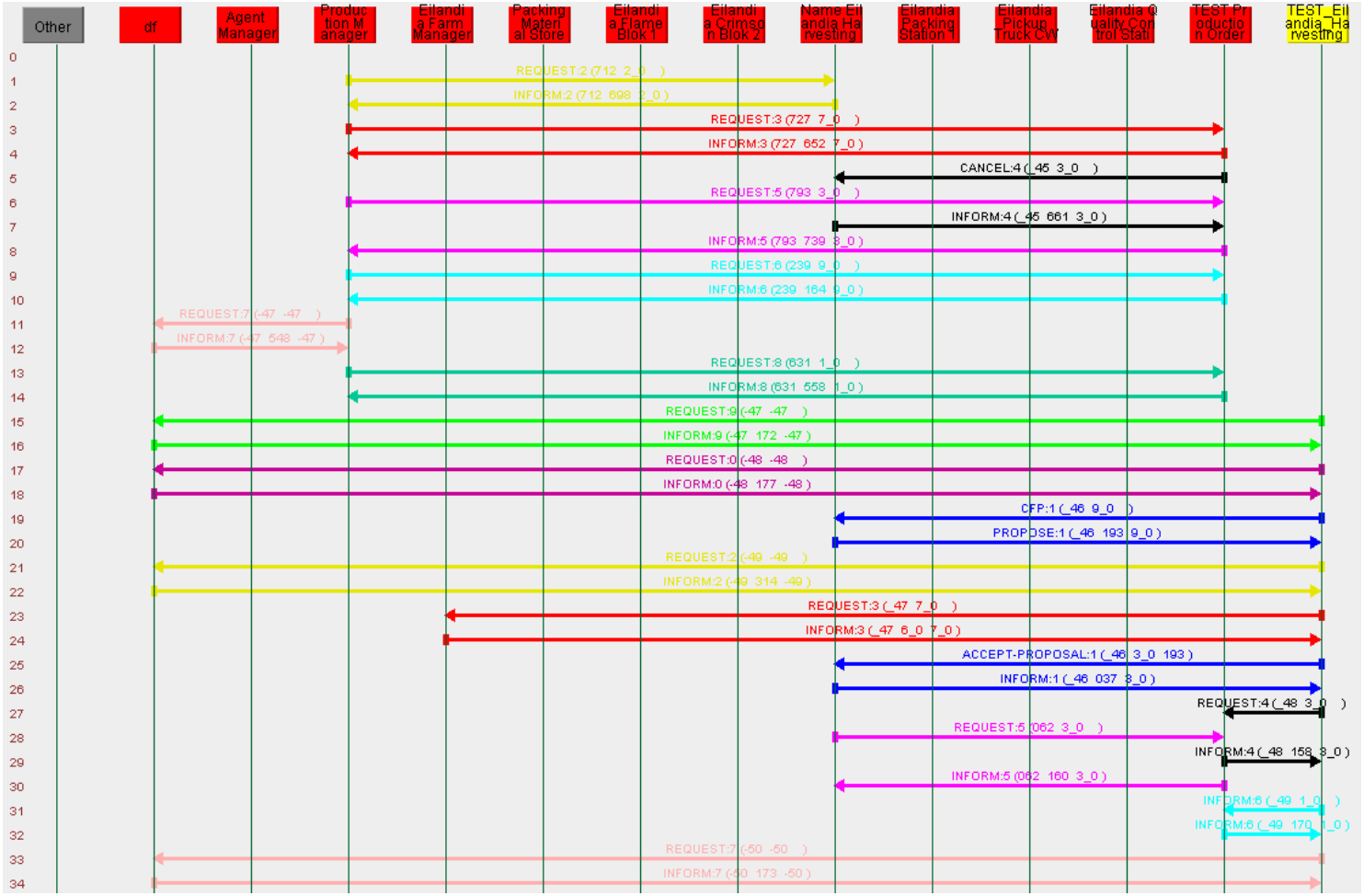


Figure 49: Replace a harvesting team resource.

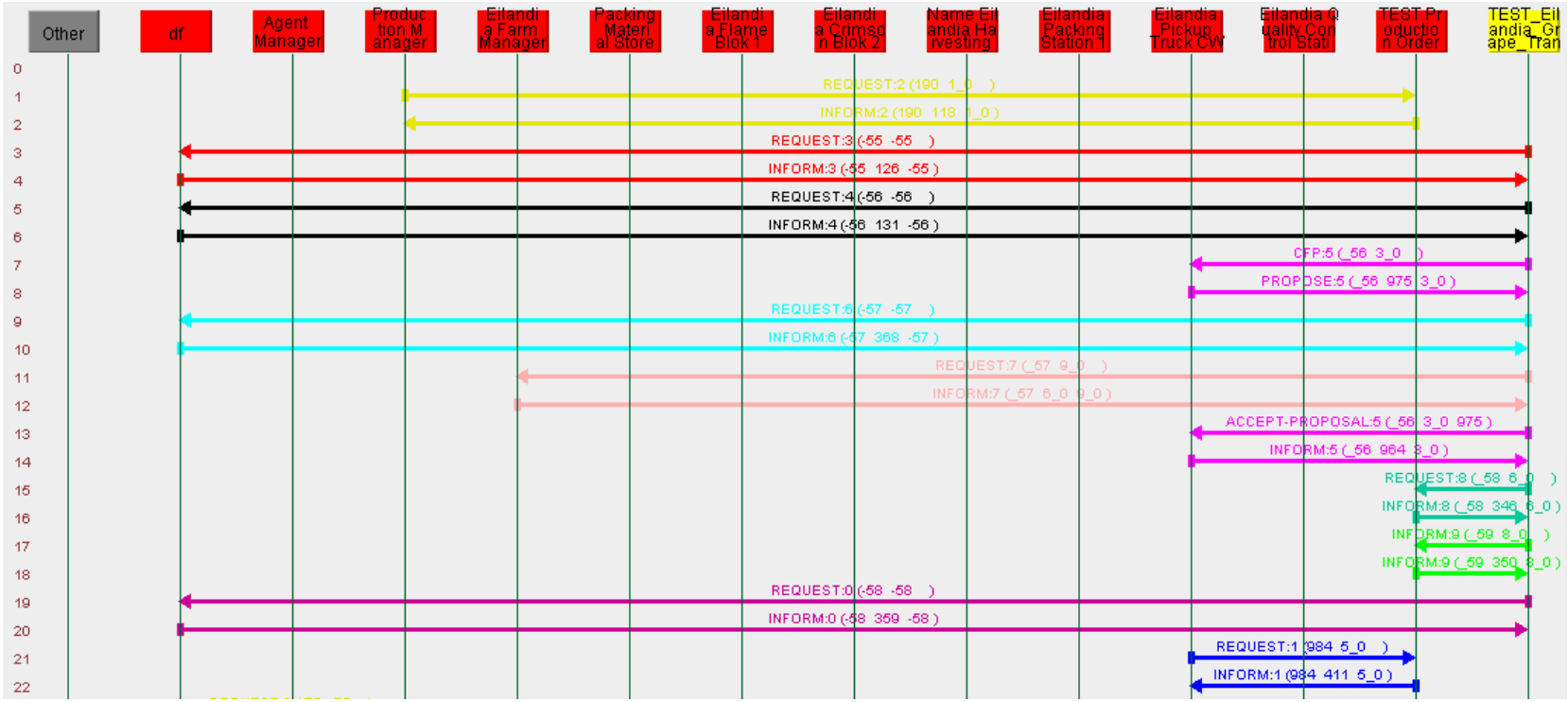


Figure 50: Add a grape transportation vehicle.

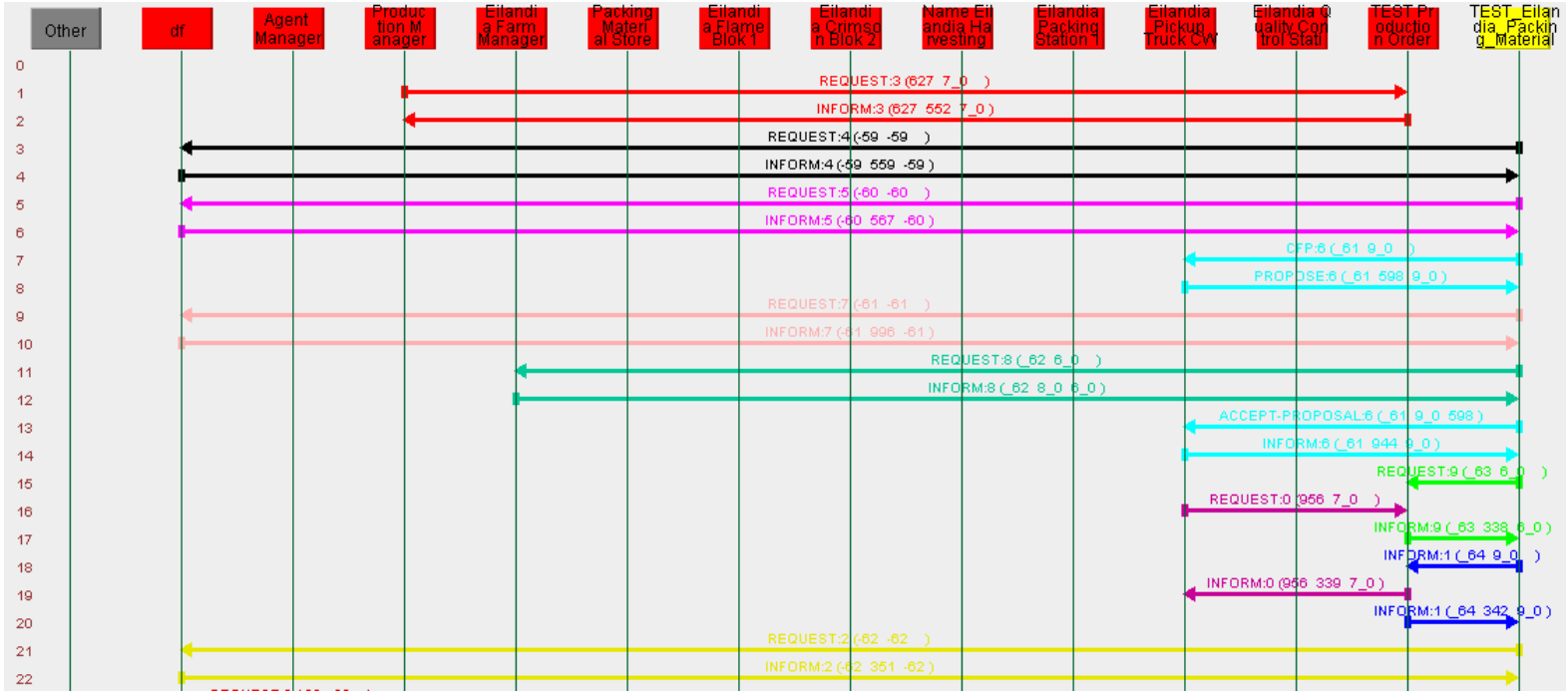


Figure 51: Add a packing material transportation vehicle.

A.5 Cancel Production Order

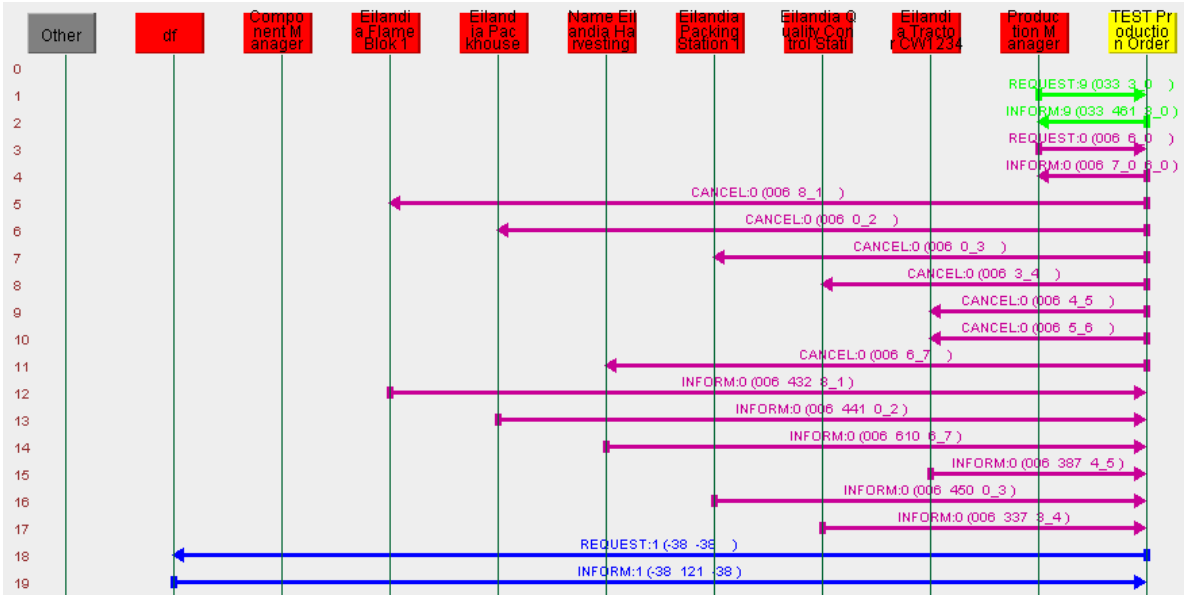


Figure 52: Cancel production order.

Appendix B Messages in XML Format

B.1 Vineyard resource proposal information message in XML format

```
<VineyardResource resourceName=\"Eilandia Flame Blok 1\">
    <vineyardVariety>Flame</vineyardVariety>
    <vineyardFarm>Eilandia</vineyardFarm>
    <vineyardLocation>Blok 1</vineyardLocation>
    <vineyardSize>2</vineyardSize>
    <vineyardCapacity>5000</vineyardCapacity>
    <vineyardBerrySize>L</vineyardBerrySize>
    <vineyardSugarLevel>8</vineyardSugarLevel>
    <vineyardBerryColour>1</vineyardBerryColour>
    <vineyardBerryBlemish>1</vineyardBerryBlemish>
    <resourceAvailability>Unassigned</resourceAvailability>
</VinevardResource>"
```

B.2 Request assigned resources inform message in XML format

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProductionOrder productionOrderName="TEST">
    <productionOrderQuantity>5</productionOrderQuantity>
    <vineyard>Eilandia Flame Blok 1</vineyard>
    <vineyard>Eilandia Crimson Blok 2</vineyard>
    <packhouses>Eilandia Packhouse</packhouses>
    <packingMaterials>Mooi 5 kg</packingMaterials>
    <packingMaterials>250 g</packingMaterials>
    <packingMaterials>Label</packingMaterials>
    <harvestingTeams>Supervisor Eilandia Harvesting Team</harvestingTeams>
    <qualityControlStation>Eilandia Quality Control Station 1</qualityControlStation>
    <packingStations>Eilandia Packing Station 1</packingStations>
    <grapeTransportationVehicles>Eilandia Tractor</grapeTransportationVehicles>
    <packingMaterialTransportationVehicles>Eilandia Truck</packingMaterialTransportationVehicles>
</ProductionOrder>
```

B.3 Stored production order information in XML format

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProductionOrder productionOrderName="TEST">
    <vineyard>Eilandia Flame Blok 1</vineyard>
    <packhouses>Eilandia Packhouse</packhouses>
    <packingMaterials>Mooi 5 kg</packingMaterials>
    <packingMaterials>250 g</packingMaterials>
    <packingMaterials>Label</packingMaterials>
    <harvestingTeams>Supervisor Eilandia Harvesting Team</harvestingTeams>
    <qualityControlStation>Eilandia Quality Control Station 1</qualityControlStation>
    <packingStations>Eilandia Packing Station 1</packingStations>
    <grapeTransportationVehicles>Eilandia Tractor</grapeTransportationVehicles>
    <packingMaterialTransportationVehicles>Eilandia Truck</packingMaterialTransportationVehicles>
</ProductionOrder>
```

Appendix C JADE Source Code

This appendix presents the source code of a vineyard resource. The vineyard resource represents a resource running on a PC. The source code shows the implementation of a resource according to the four ARTI components used throughout this thesis.

C.1 Vineyard Resource

C.1.1 RTIB Component

```
public class Vineyard_IBRT extends Agent {

	// ----- Agent variables ---------------------------------------------
	// Intelligent Agent Activity Type Object
	Vineyard_IART vineyardIART = new Vineyard_IART();
	// Vector to keep resource schedule
	Vector<String> vineyardResourceSchedule = new Vector<String>();
	// -------------------------------------------------------------------

	/*
	 * Build and return the FSM Behaviour to execute
	 */
	public FSMBehaviour getFSMBehaviour(Object arguments) {
		// Create FSMBehaviour
		FSMBehaviour fsmBehaviour = new FSMBehaviour();

		// Set DataStore
		initiateHandleResourceAssignemtRequests.setDataStore(fsmBehaviour.getDataStore());
		SSResponderDispatcherBehaviour.setDataStore(fsmBehaviour.getDataStore());
```

```
            // Register the behaviours in the FSM States
            fsmBehaviour.registerFirstState(initiateHandleResourceAssignemtRequests,
"initiateHandleResourceAssignemtRequests");
            fsmBehaviour.registerState(SSResponderDispatcherBehaviour, "SSResponderDispatcherBehaviour");
            fsmBehaviour.registerLastState(terminateActivityInstance, "terminateActivityInstance");

            // Register transitions in the FSM
            fsmBehaviour.registerTransition("initiateHandleResourceAssignemtRequests", "terminateActivityInstance",
0);
            fsmBehaviour.registerTransition("initiateHandleResourceAssignemtRequests",
"SSResponderDispatcherBehaviour", 1);
            fsmBehaviour.registerTransition("SSResponderDispatcherBehaviour", "terminateActivityInstance", 2);

            // Return the FSMBehaviour to be executed by the Activity Instance
            return fsmBehaviour;
      }

      // ----------------------------------------------------------------------
      // Build the xml string with proposal messages
      // ----------------------------------------------------------------------
      public String buildVineyardXML(String vineyardLocalName, String vineyardVariety, String vineyardFarm, String
vineyardLocation, String vineyardSize, String vineyardCapacity, String vineyardBerrySize, String vineyardSugarLevel,
String vineyardBerryColour, String vineyardBerryBlemish,Vector<String> vineyardSchedule) {
            String xmlString = null;

            // Build the XML Object
            Vineyard_XML vineyardXML = new Vineyard_XML();
            vineyardXML.setResourceName(vineyardLocalName);
            vineyardXML.setVineyardVariety(vineyardVariety);
            vineyardXML.setVineyardFarm(vineyardFarm);
            vineyardXML.setVineyardLocation(vineyardLocation);
            vineyardXML.setVineyardSize(vineyardSize);
            vineyardXML.setVineyardCapacity(vineyardCapacity);
```

```
        vineyardXML.setVineyardBerrySize(vineyardBerrySize);
        vineyardXML.setVineyardSugarLevel(vineyardSugarLevel);
        vineyardXML.setVineyardBerryColour(vineyardBerryColour);
        vineyardXML.setVineyardBerryBlemish(vineyardBerryBlemish);
        if(vineyardSchedule.size() == 0) {
              Vector<String> resourceUnassigned = new Vector<String>();
              resourceUnassigned.addElement("Unassigned");
              vineyardXML.setResourceAvailability(resourceUnassigned);
        } else {
              vineyardXML.setResourceAvailability(vineyardSchedule);
        }

        // Create the XML String from the XML Object
        try {
              // Creating the JAXB context
              JAXBContext jaxbContext = JAXBContext.newInstance(Vineyard_XML.class);
              // Creating the marshaller object
              Marshaller marshallObject = jaxbContext.createMarshaller();
              // Setting the property to show xml format output
              marshallObject.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);

              // Write to String
              StringWriter stringWriterObject = new StringWriter();
              marshallObject.marshal(vineyardXML, stringWriterObject);
              xmlString = stringWriterObject.toString();

        } catch (JAXBException e) {
              e.printStackTrace();
        }


        return xmlString;
}
```

```
// ----------------------------------------------------------------------
// Parse the xml string to a xml object
// ----------------------------------------------------------------------
public Vineyard_XML parseVineyardXMLMessages(String posposalMessagesXMLString) {
	// XML Object
	Vineyard_XML XMLFormatObject = null;
	// Parse the String to a XML Object
	try {
		//Creating the JAXB context
		JAXBContext jaxbContext = JAXBContext.newInstance(Vineyard_XML.class);
		// Creating the unmarshaller object
		Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

		StringReader reader = new StringReader(posposalMessagesXMLString);
		XMLFormatObject = (Vineyard_XML) unmarshaller.unmarshal(reader);
	} catch (JAXBException e) {
		e.printStackTrace();
	}
	return XMLFormatObject;
}

//=============================================================================================
//		OneShotBehaviour to initiate ContractNetResponder and SSResponderDispatcher
//=============================================================================================
public OneShotBehaviour initiateHandleResourceAssignemtRequests = new OneShotBehaviour(this) {
	public void action() {
		HandleResourceAssignemtRequests.setDataStore(getParent().getDataStore());
		myAgent.addBehaviour(HandleResourceAssignemtRequests);
	}
	public int onEnd() {
		return vineyardIART.getNextBehaviour("initiateHandleResourceAssignemtRequests");
	}
};
```

```
	//============================================================================================
	//                        ContractNetResponder to handle resource assignment requests
	//============================================================================================
	public ContractNetResponder HandleResourceAssignemtRequests = new ContractNetResponder(this,
MessageTemplate.MatchPerformative(ACLMessage.CFP)) {

		@SuppressWarnings("unchecked")
		protected ACLMessage handleCfp(ACLMessage cfp) {
			// Get vineyard information
			String vineyardLocalName = (String) getDataStore().get("LOCALNAME");
			String vineyardVariety = (String) getDataStore().get("VINEYARD_VARIETY");
			String vineyardFarm = (String) getDataStore().get("VINEYARD_FARM");
			String vineyardLocation = (String) getDataStore().get("VINEYARD_LOCATION");
			String vineyardSize = (String) getDataStore().get("VINEYARD_SIZE");
			String vineyardCapacity = (String) getDataStore().get("VINEYARD_CAPACITY");
			String vineyardBerrySize = (String) getDataStore().get("VINEYARD_BERRY_SIZE");
			String vineyardSugarLevel = (String) getDataStore().get("VINEYARD_SUGAR_LEVEL");
			String vineyardBerryColour = (String) getDataStore().get("VINEYARD_BERRY_COLOUR");
			String vineyardBerryBlemish = (String) getDataStore().get("VINEYARD_BERRY_BLEMISH");
			Vector<String> vineyardSchedule = (Vector<String>) getDataStore().get("VINEYARD_SCHEDULE");

			// Create XML of the resource's information
			String resourceInformationString = buildVineyardXML(vineyardLocalName, vineyardVariety,
vineyardFarm, vineyardLocation, vineyardSize, vineyardCapacity, vineyardBerrySize, vineyardSugarLevel,
vineyardBerryColour, vineyardBerryBlemish, vineyardSchedule);

			// Create the proposal message
			ACLMessage cfpReply = new ACLMessage(ACLMessage.PROPOSE);
			cfpReply.setContent(resourceInformationString);
			cfpReply.addReceiver(cfp.getSender());

			return cfpReply;
		}
```

```
        // Handles the ACCEPT_PROPOSAL message - send INFORM or FAILURE
        @SuppressWarnings("unchecked")
        protected ACLMessage handleAcceptProposal(ACLMessage cfp, ACLMessage propose, ACLMessage accept) {
                String vineyardFarm = (String) getDataStore().get("VINEYARD_FARM");

                ACLMessage msgInform = accept.createReply();
                msgInform.setPerformative(ACLMessage.INFORM);
                msgInform.setContent(vineyardFarm);

                // Update the resource schedule vector
                Vector<String> resourceScheduleVector = (Vector<String>) getDataStore().get("VINEYARD_SCHEDULE");
                resourceScheduleVector.addElement(accept.getContent());
                vineyardResourceSchedule = resourceScheduleVector;
                getDataStore().put("VINEYARD_SCHEDULE", vineyardResourceSchedule);

                return msgInform;
        }

        protected void handleOutOfSequence(ACLMessage msg) {
                ACLMessage msgReply = new ACLMessage(ACLMessage.INFORM);
                msgReply.createReply();

                getDataStore().put(msgReply, REPLY_KEY);
        }

        public int onEnd() {
                return vineyardIART.getNextBehaviour("HandleResourceAssignemtRequests");
        }

};
```

```
	//=============================================================================================
	//		OneShotBehaviour to Terminate this Activity Instance Agent
	//=============================================================================================
	public OneShotBehaviour terminateActivityInstance = new OneShotBehaviour() {
		public void action() {
			// Terminates the thread
			myAgent.doDelete();
		}
	};

	//=============================================================================================
	//		SSResponderDispatcher to handle all request messages
	//=============================================================================================
	MessageTemplate msgTemp = MessageTemplate.or(MessageTemplate.MatchPerformative(ACLMessage.REQUEST),
MessageTemplate.MatchPerformative(ACLMessage.CANCEL));
	public SSResponderDispatcher SSResponderDispatcherBehaviour = new SSResponderDispatcher(this, msgTemp) {

		// Responder Behaviour to execute on received request message
		SSIteratedAchieveREResponder responderBehaviour = null;

		/*
		 * Return the behaviour to execute
		 */
		protected Behaviour createResponder(ACLMessage msgRequest) {

			// Variables
			DataStore responderBehaviourDataStore = getParent().getDataStore();

			// ----- Create possible executable behaviours -----------------------------------------
			// Handle request to terminate the Agent
			SSIteratedAchieveREResponder TerminateAgent = new SSIteratedAchieveREResponder(myAgent,
msgRequest) {
```

```
                        // Method to handle all request messages
                        protected ACLMessage handleRequest(ACLMessage request) {

                                ACLMessage msgResponse = request.createReply();
                                msgResponse.setPerformative(ACLMessage.INFORM);
                                msgResponse.setContent("Agent Terminating ...");

                                return msgResponse;
                        }

                        public int onEnd() {
                                // Terminates the thread
                                myAgent.doDelete();
                                return 0;
                        };
                };

                // Handle request to cancel a production order
                SSIteratedAchieveREResponder HandleCancellationMessages = new
SSIteratedAchieveREResponder(myAgent, msgRequest) {

                        // Handle the request to cancel production order
                        @SuppressWarnings("unchecked")
                        protected ACLMessage handleRequest(ACLMessage msgRequest) {
                                // Vector containing the resource's schedule
                                Vector<String> resourceScheduleVector = (Vector<String>)
getDataStore().get("resourceSchedule");
                                // Remove the element from the resource's schedule
                                vineyardResourceSchedule.remove(msgRequest.getContent());
                                // Put the schedule vector back in to resource datastore
                                getDataStore().put("VINEYARD_SCHEDULE", resourceScheduleVector);

                                ACLMessage msgResponse = msgRequest.createReply();
```

```
                                msgResponse.setPerformative(ACLMessage.INFORM);
                                msgResponse.setOntology("Production Order Removed");
                                msgResponse.setContent(myAgent.getLocalName());

                                return msgResponse;
                        }

                };

                // Handle request farm
                SSIteratedAchieveREResponder HandleFarmRequestMessages = new
SSIteratedAchieveREResponder(myAgent, msgRequest) {

                        // Handle the request to cancel production order
                        protected ACLMessage handleRequest(ACLMessage msgRequest) {
                                // Create vector string for message content
                                Vector<String> msgContent = new Vector<String>();
                                msgContent.addElement((String) responderBehaviourDataStore.get("VINEYARD_FARM"));
        // Resource Farm
                                msgContent.addElement("Vineyard_Resource");
                        // Resource Type
                                // Create the reply message
                                ACLMessage msgResponse = msgRequest.createReply();
                                msgResponse.setPerformative(ACLMessage.INFORM);
                                msgResponse.setContent(msgContent.toString());
                                // Send reply message
                                return msgResponse;
                        }

                };
```

```
                    // Handle request to update data
                    SSIteratedAchieveREResponder HandleDataUpdateMessages = new SSIteratedAchieveREResponder(myAgent,
msgRequest) {
                          protected ACLMessage handleRequest(ACLMessage request) {
                                // Get information from message
                                Vector<String> updateInformationVector =
convertStringToVector(msgRequest.getContent());
                                String informationType = updateInformationVector.get(0);
                                String value = updateInformationVector.get(1);

                                // Update datastore information
                                responderBehaviourDataStore.put(informationType, value);

                                // Send Reply
                                ACLMessage msgResponse = request.createReply();
                                msgResponse.setPerformative(ACLMessage.INFORM);
                                msgResponse.setContent("Data Updated");

                                return msgResponse;
                          };

                          protected Vector<String> convertStringToVector(String contentVectorString) {
                                // Vector that will contain selected resources
                                Vector<String> contentVector = new Vector<String>();
                                // Convert String to Chars
                                char[] StringChars = contentVectorString.toCharArray();
                                // Step through Chars to build Strings and Vector
                                int index = 0;
                                String stringChars = "";
                                for(int i = 0; i < contentVectorString.length(); ++i) {
                                      if(StringChars[i] == '[') {
                                            i++;
                                      } if(StringChars[i] == ',' || StringChars[i] == ']') {
```

```
                                        contentVector.add(index, stringChars);
                                        i++; index++; stringChars = "";
                                  } else {
                                        stringChars = stringChars + String.valueOf(StringChars[i]);
                                  }
                            }

                            return contentVector;
                      }
                };

                // Handle data requests
                SSIteratedAchieveREResponder HandleVineyardDataRequestMessage = new
SSIteratedAchieveREResponder(myAgent, msgRequest) {
                      @SuppressWarnings("unchecked")
                      protected ACLMessage handleRequest(ACLMessage request) {
                            // Get vineyard information
                            String vineyardLocalName = (String) responderBehaviourDataStore.get("LOCALNAME");
                            String vineyardVariety = (String)
responderBehaviourDataStore.get("VINEYARD_VARIETY");
                            String vineyardFarm = (String) responderBehaviourDataStore.get("VINEYARD_FARM");
                            String vineyardLocation = (String)
responderBehaviourDataStore.get("VINEYARD_LOCATION");
                            String vineyardSize = (String) responderBehaviourDataStore.get("VINEYARD_SIZE");
                            String vineyardCapacity = (String)
responderBehaviourDataStore.get("VINEYARD_CAPACITY");
                            String vineyardBerrySize = (String)
responderBehaviourDataStore.get("VINEYARD_BERRY_SIZE");
                            String vineyardSugarLevel = (String)
responderBehaviourDataStore.get("VINEYARD_SUGAR_LEVEL");
                            String vineyardBerryColour = (String)
responderBehaviourDataStore.get("VINEYARD_BERRY_COLOUR");
```

```
                                String vineyardBerryBlemish = (String)
responderBehaviourDataStore.get("VINEYARD_BERRY_BLEMISH");
                                Vector<String> vineyardSchedule = (Vector<String>)
responderBehaviourDataStore.get("VINEYARD_SCHEDULE");

                                // Create XML String of the resource's information
                                String xmlString = buildVineyardXML(vineyardLocalName, vineyardVariety,
vineyardFarm, vineyardLocation, vineyardSize, vineyardCapacity, vineyardBerrySize, vineyardSugarLevel,
vineyardBerryColour, vineyardBerryBlemish, vineyardSchedule);

                                // Create the reply message
                                ACLMessage msgResponse = msgRequest.createReply();
                                msgResponse.setPerformative(ACLMessage.INFORM);
                                msgResponse.setOntology("Vineyard");
                                msgResponse.setContent(xmlString);

                                // Send reply message
                                return msgResponse;
                        };
                };
                // -----------------------------------------------------------------------------------------

                // Vector containing executable behaviour for production order IAAT to choose from
                Vector<SSIteratedAchieveREResponder> executableBehaviours = new
Vector<SSIteratedAchieveREResponder>();
                executableBehaviours.addElement(TerminateAgent);
                executableBehaviours.addElement(HandleCancellationMessages);
                executableBehaviours.addElement(HandleFarmRequestMessages);
                executableBehaviours.addElement(HandleDataUpdateMessages);
                executableBehaviours.addElement(HandleVineyardDataRequestMessage);
```

```
                // Get vector index of behaviour to execute
                int executableBehavioursIndex = vineyardIART.getNextResponderBehaviour(msgRequest);
                // Get the behaviour to execute from vector using index
                responderBehaviour = executableBehaviours.get(executableBehavioursIndex);
                // Close/Terminate behaviour when the current session ends
                responderBehaviour.closeSessionOnNextReply();

                // Return selected behaviour for execution
                return responderBehaviour;
        }

        public int onEnd() {
                return vineyardIART.getNextBehaviour("SSResponderDispatcherBehaviour");
        }

};


}
```

C.1.2 RTIA Componet

```
// Fills the datastore of the activity instance with the arguments received
public DataStore buildDatastore(Object[] arguments) {
        DataStore vineyardDatastore = new DataStore();
        // Add the resource's information from arguments
        Vector<String> resourceInfoVector= new Vector<String>();
        resourceInfoVector.addElement((String) arguments[1]);      // localName
        resourceInfoVector.addElement((String) arguments[4]);      // vineyardVariety
        resourceInfoVector.addElement((String) arguments[3]);      // vineyardResourceFarm
        resourceInfoVector.addElement((String) arguments[5]);      // vineyardLocation
        resourceInfoVector.addElement((String) arguments[6]);      // vineyardSize
        resourceInfoVector.addElement((String) arguments[7]);      // vineyardCapacity
```

```
        resourceInfoVector.addElement((String) arguments[8]);      // vineyardBerrySize
        resourceInfoVector.addElement((String) arguments[9]);      // vineyardSugarLevel
        resourceInfoVector.addElement((String) arguments[10]);     // vineyardBerryColour
        resourceInfoVector.addElement((String) arguments[11]);     // vineyardBerryBlemish
        vineyardDatastore.put("resourceInformation", resourceInfoVector);

        // Build DataStore
        vineyardDatastore.put("LOCALNAME", (String) arguments[1]);
        vineyardDatastore.put("VINEYARD_VARIETY", (String) arguments[4]);
        vineyardDatastore.put("VINEYARD_FARM", (String) arguments[3]);
        vineyardDatastore.put("VINEYARD_LOCATION", (String) arguments[5]);
        vineyardDatastore.put("VINEYARD_SIZE", (String) arguments[6]);
        vineyardDatastore.put("VINEYARD_CAPACITY", (String) arguments[7]);
        vineyardDatastore.put("VINEYARD_BERRY_SIZE", (String) arguments[8]);
        vineyardDatastore.put("VINEYARD_SUGAR_LEVEL", (String) arguments[9]);
        vineyardDatastore.put("VINEYARD_BERRY_COLOUR", (String) arguments[10]);
        vineyardDatastore.put("VINEYARD_BERRY_BLEMISH", (String) arguments[11]);
        // Add vector to foRm resource schedule
        Vector<String> resourceScheduleVector = new Vector<String>();
        vineyardDatastore.put("VINEYARD_SCHEDULE", resourceScheduleVector);

        return vineyardDatastore;
}


// Provides the information of next behaviour to intelligent being activity instance
public String getFSMBehaviour() {
        return "intelligent_being_resource_type.Vineyard_IBRT";
}
```

```
// Return event integer to OnEnd for FSM to execute next behaviour
public int getNextBehaviour(String currentbehaviour) {
	int nextBehaviour = 0;

	switch(currentbehaviour) {
	case "initiateHandleResourceAssignemtRequests":
		nextBehaviour = 1;
		break;
	case "HandleResourceAssignemtRequests":
		nextBehaviour = 1;
		break;
	case "SSResponderDispatcherBehaviour":
		nextBehaviour = 2;
		break;
	}

	return nextBehaviour;
}


// Get the behaviours vector index of next behaviour to execute
public int getNextResponderBehaviour(ACLMessage msgRequest) {

	int behaviourIndex = 0;

	int msgPerformative = msgRequest.getPerformative();

	switch(msgPerformative) {
	case ACLMessage.CANCEL:
		behaviourIndex = 1;
		break;
```

```
        case ACLMessage.REQUEST:
                switch(msgRequest.getOntology()) {
                case "RESOURCE_UPDATE":
                        behaviourIndex = 2;
                        break;
                case "DATA_UPDATE":
                        behaviourIndex = 3;
                        break;
                case "REQUEST_DATA":
                        behaviourIndex = 4;
                        break;
                case "TERMINATE":
                        behaviourIndex = 0;
                        break;
                }
                break;
        default:
                behaviourIndex = 0;
                System.out.println("\n"
                                + "=====================================\n"
                                + "Resource Received Message To Terminate\n"
                                + "=====================================\n");
                break;
        }

        return behaviourIndex;
}
```

C.1.3 RIIA Component

```
// Intelligent Agent Resource Type Objects
Vineyard_IART vineyardIART = new Vineyard_IART();
Packhouse_IART packhouseIART = new Packhouse_IART();

/*
 * Fill datastore with received arguments
 */
public DataStore handleArguments(String agentType, Object[] arguments) {
	DataStore resourceInstanceDatastore = new DataStore();

	switch(agentType) {
	case "Vineyard_Resource":
		resourceInstanceDatastore = vineyardIART.buildDatastore(arguments);
		break;
	case "Packhouse_Resource":
		resourceInstanceDatastore = packhouseIART.buildDatastore(arguments);
		break;
	}

	return resourceInstanceDatastore;
}


/*
 * Get the class name of the intelligent being resource type
 */
public String getFSMBehaviour(String agentType) {
	String fsmClassName = null;
	switch(agentType) {
	case "Vineyard_Resource":
		fsmClassName = vineyardIART.getFSMBehaviour();
		break;
```

```
		case "Packhouse_Resource":
			fsmClassName = packhouseIART.getFSMBehaviour();
			break;
		}

		return fsmClassName;
	}
```

C.1.4 RIIB Component

```
public class Resource_Instance_Intelligent_Being extends Agent {

	// ----- Agent variables ---------------------------------------------------
	// Intelligent Agent Resource Instance Object
	Intelligent_Agent_Resource_Instance intelligentAgentResourceInstance = new Intelligent_Agent_Resource_Instance();
	// Arguments
	Object arguments[];
	String agentType;
	String serviceDescription;
	// Behaviours
	String currentBehaviour = "First Behaviour";
	// Datastore
	DataStore resourceInstanceDatastore = new DataStore();
	// -------------------------------------------------------------------------

	/*
	 * Agent Setup
	 */
	protected void setup() {
		// ------- Get arguments -----------------------------------------------
		arguments = getArguments();
		agentType = (String) arguments[0];
		serviceDescription = (String) arguments[2];
		resourceInstanceDatastore = intelligentAgentResourceInstance.handleArguments(agentType, arguments);
		// ---------------------------------------------------------------------
```

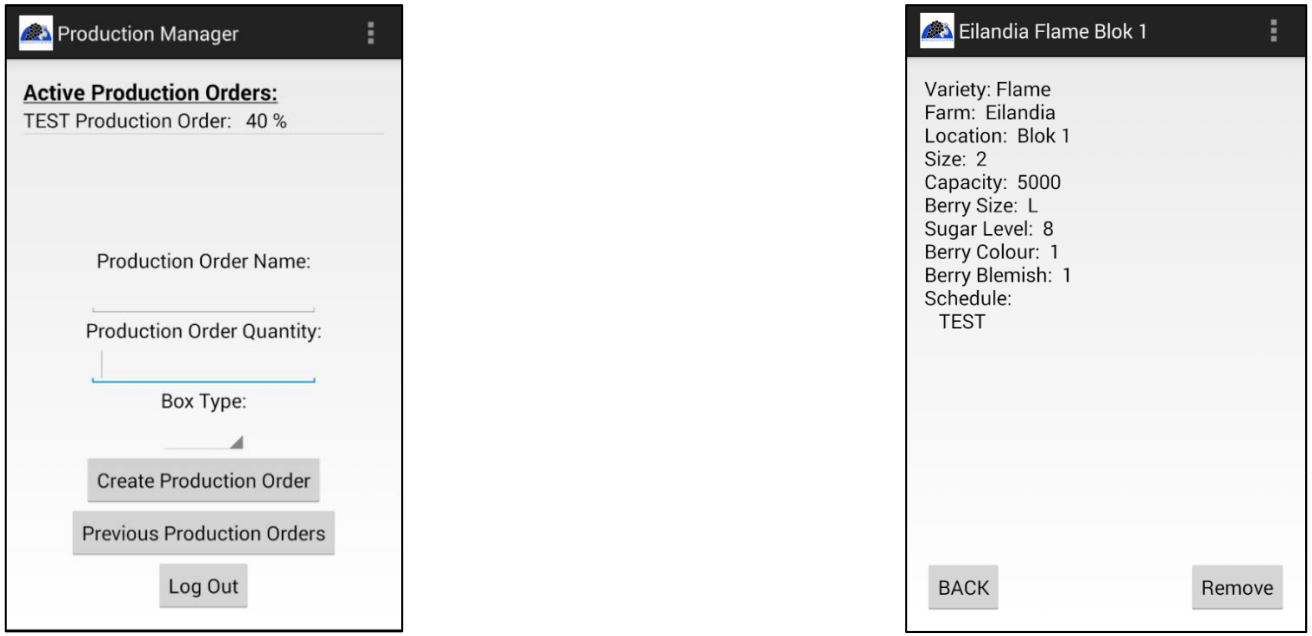
```
        // ------- Register the Agent service in the yellow pages -----------------
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setType(serviceDescription);
        sd.setName(getAID().getLocalName());
        dfd.addServices(sd);
        try {
                DFService.register(this, dfd);
        }
        catch (FIPAException fe) {
                fe.printStackTrace();
        }
        // ------------------------------------------------------------------------

        // ------- Add Behaviours -------------------------------------------------
        resourceInstanceBehaviour.setDataStore(resourceInstanceDatastore);
        addBehaviour(resourceInstanceBehaviour);
        // ------------------------------------------------------------------------
}


/*
 * Terminates the agent
 */
protected void takeDown() {
        // Deregister from the yellow pages
        try {
                DFService.deregister(this);
        }
        catch (FIPAException fe) {
                fe.printStackTrace();
        }
        System.out.println(getLocalName() +" terminated.");
}
```

```
	//=============================================================================================
	//		OneShotBehaviour to execute the correct FSMBEhaviour
	//=============================================================================================
	@SuppressWarnings("rawtypes")
	OneShotBehaviour resourceInstanceBehaviour = new OneShotBehaviour() {

		// Variables
		String className;
		Class classRef;

		@SuppressWarnings("unchecked")
		public void action() {
			// Get class name of intelligent being resource type
			className = intelligentAgentResourceInstance.getFSMBehaviour(agentType);

			// Execute the methodName in the className
			try {
				// Get the class
				classRef = Class.forName(className);
				// Create an instance of the class with className
				Object instance = classRef.newInstance();
				// Get the FSMBehaviour to execute from the methodName
				Class<Object> parameterTypesArray = Object.class;
				Method method = classRef.getDeclaredMethod("getFSMBehaviour", parameterTypesArray);
				// Invoke an instance of the method to get behaviour
				Object args = arguments;
				FSMBehaviour fsmBehaviour =  (FSMBehaviour) method.invoke(instance, args);
				// Provide Behaviour with necessary information through the datastore
				fsmBehaviour.setDataStore(getDataStore());
				// Execute the behaviour from activity instance thread
				myAgent.addBehaviour(fsmBehaviour);
			} catch (ClassNotFoundException | InstantiationException | IllegalAccessException | NoSuchMethodException |
SecurityException | IllegalArgumentException | InvocationTargetException e) {
				e.printStackTrace();
			}
		}

	};
```

Appendix D Android Application Screenshots

D.1 Production Manager

Figure 53 (a) shows the home screen of a production manager resource with the active production orders and their progress. Figure 53 (b) shows the information which the production manager can obtain from an assigned vineyard resource.



(a) (b)

Figure 53: Production manager home screen (a) and production manager screen to view a vineyard resource (b).

Figure 54 (a) shows the screen where the production manager needs to choose vineyard resources to assign to a production order. Figure 54 (b) shows the vineyard resources and their information when the production manager wants to add an additional resource.

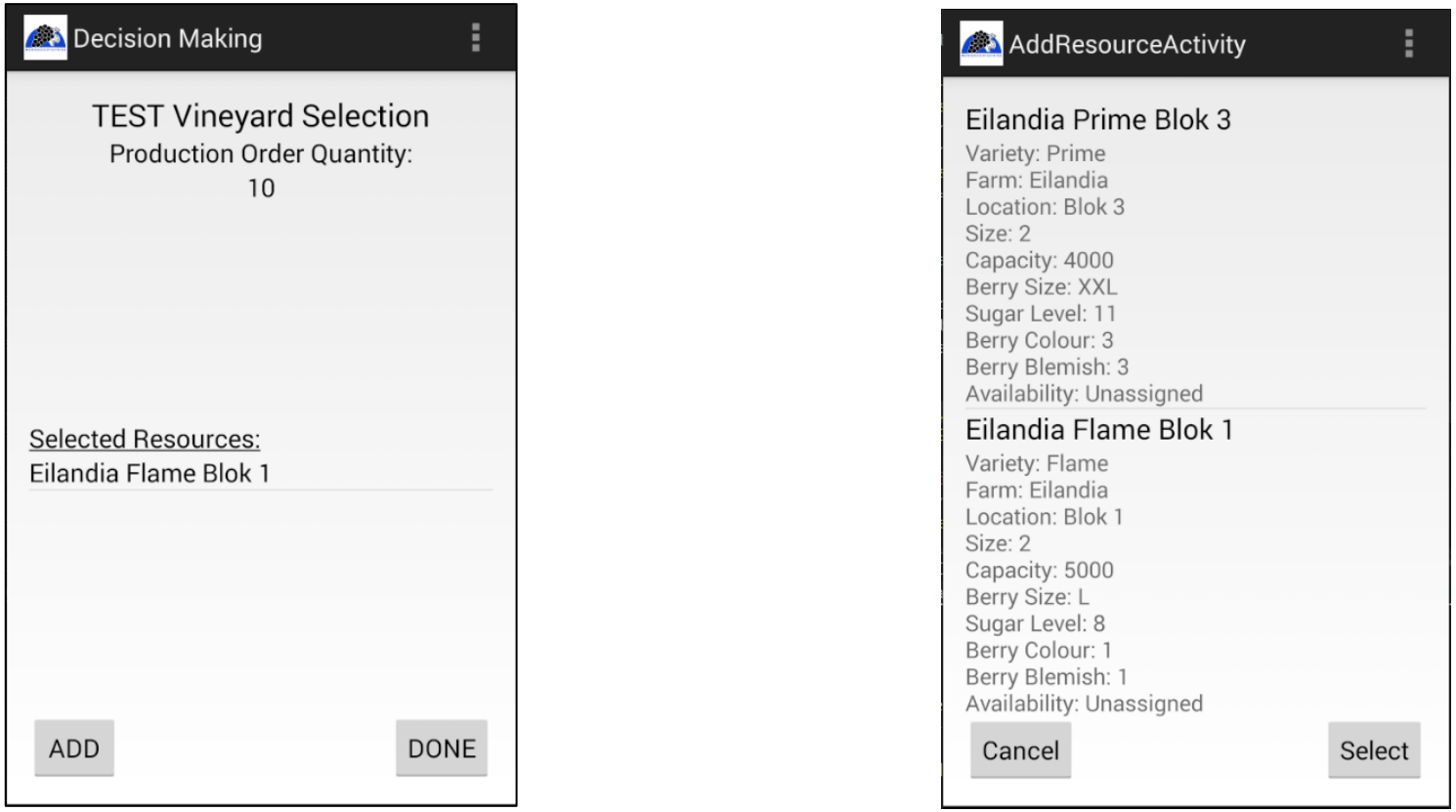
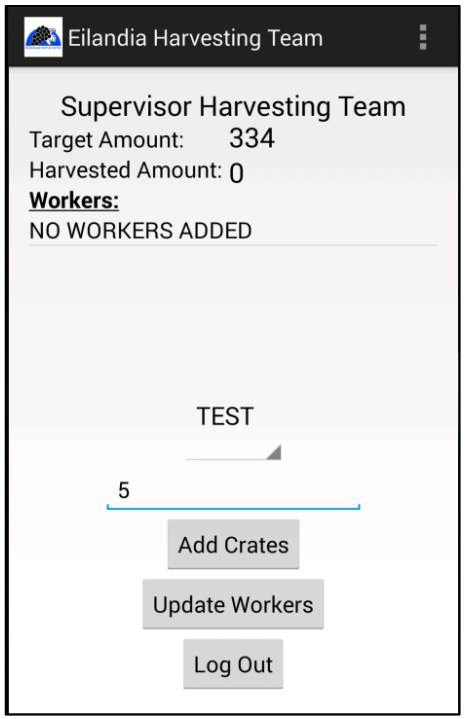


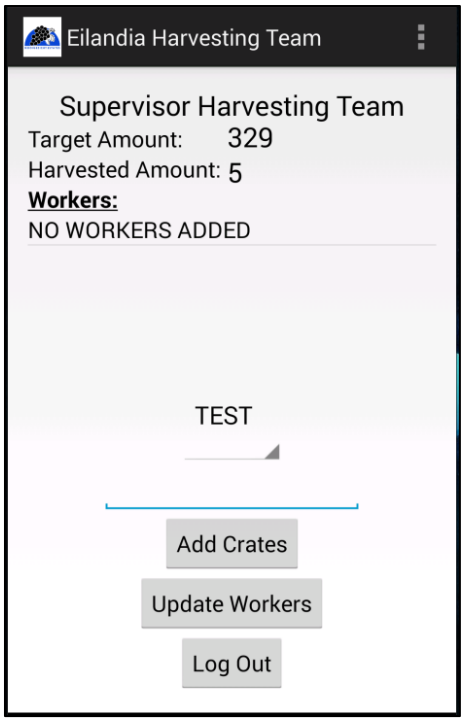
Figure 54: Production manager resource selection screen (a) and production manager resource addition screen (b).

D.2 Resources

Figure 55 (a) show the screen of a harvesting team resource before adding 5 harvested crates. Figure 55 (b) show the screen of the same harvesting team after 5 crates have been added. The target value and harvested crates for the TEST production order are visible in both figures.



(a)



(b)

Figure 55: Harvesting team resource before adding 5 crates to the TEST production order (a) and after adding 5 crates (b).